Karl E. Kurbel

The Making of Information Systems

Software Engineering and Management in a Globalized World

Springer

The Making of Information Systems

Karl E. Kurbel

The Making of Information Systems

Software Engineering and Management in a Globalized World

Springer

Karl Kurbel holds the Chair of Business Informatics at the European University Viadrina Frankfurt (Oder) in Germany

Prof. Dr. Karl E. Kurbel
European University Viadrina Frankfurt (Oder)
Chair of Business Informatics
15230 Frankfurt (Oder), Germany
kurbel.BI@uni-ffo.de

Chapter 9 was contributed by Brandon Ulrich, Managing Director of B2 International, a Budapest, Hungary, based software company

Brandon Ulrich
B2 International Ltd.
Madach ter 3, Em. III
1075 Budapest, Hungary
bulrich@b2international.com

ISBN: 978-3-540-79260-4 e-ISBN: 978-3-540-79261-1

DOI: 10.1007/978-3-540-79261-1

Library of Congress Control Number: 2008925552

Cover design: WMX Design GmbH, Heidelberg

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

Information systems (IS) are the backbone of any organization today. Practically all major business processes and business functions are supported by information systems. It is inconceivable that a business firm – or any other non-trivial organization – would be able to operate without powerful information systems.

This book deals with the question: Where do these information systems come from? In previous decades, the answer seemed fairly obvious: An organization would have their IT (information technology) group, and these people would develop information systems when the need arose. Most of the early books on business information systems started from this premise.

While inhouse IS development still has its role in large organizations, the number of options to obtain an information system has significantly grown. For example, an organization may choose to contract an external partner for the development. They may outsource their complete information systems development, or even their entire IT department, to such a partner. The partner can be located onshore or offshore. Many organizations establish captive centers, or they collaborate with offshore software companies in India, South America and Eastern Europe. Managing projects with globally distributed partners today creates additional challenges for the making of information systems.

Another significant change is that a good deal of large-scale information systems development (ISD) has moved from organizations whose core business *is not* software to those whose business *is* software. Fewer companies than previously actually develop individual information systems any more. In the business domain, large software vendors such as SAP, Oracle and Microsoft are providing standard software that already solves large portions of the problems for which individual information systems were developed before.

Since standard software never meets an individual organization's requirements one hundred percent, the customization of this software and its implementation in the organization have become major challenges. This means that much of the effort, time and money spent previously on information systems development now goes into customizing the standard software and adapting the organization to the software.

Taking into consideration an increasing number of already existing information systems, most organizations are facing the problem that any new system needs to be integrated into the existing IS landscape so that it smoothly collaborates with the other systems.

With the aforementioned factors in mind, this book examines and discusses the question of how information systems come into existence today. Chapter 1 describes typical information systems in a modern enterprise and which options in the making of information systems an organization faces. Chapter 2 discusses management issues and decisions regarding the launching of a project, inhouse or external development, outsourcing, offshoring as well as the costs and benefits of information systems.

The information systems architectures and platforms presented in chapter 3 play a pivotal role today. Since a new information system will most likely need to fit with the existing IS, an architecture may either be prescribed or to some extent need to be developed within the project. Flexible architectures have recently received much attention with the emergence of the SOA (service-oriented architecture) approach. Platforms provide the infrastructure for developing and running information systems.

In the fourth chapter, process models for information systems development are presented. Our investigation starts with the waterfall model and passes on to evolutionary development, prototyping, RUP (Rational unified process) and agile methodologies such as XP (extreme programming). Special attention is paid to the needs of off-shoring projects.

Chapter 5 focuses on two of the major stages in any development effort: analysis and design. Hardly any other area has received as much attention in research and practice as analysis and design. A vast body of methods and tools are available. We focus on the essential tasks to be solved in these stages, on modeling with the help of UML (unified modeling language) and on tools supporting the analysis and design activities.

In chapter 6, two more important stages are discussed: implementation and testing. In today's information systems development, these stages are largely based on automated tools such as IDEs (integrated development environments), program libraries and testing tools. Therefore we discuss not only principles and methods but also typical tool support.

Chapter 7 covers the problem of selecting, customizing and implementing standard software in an organization. Since this standard software must collaborate with other software in one way or another, integration issues are discussed in a section on EAI (enterprise application integration). A particular problem in this context is integration with so-called legacy software – i.e. information systems that are not based on current software technology but must nevertheless continue to operate.

Like many business efforts, the making of an information system is usually done in the form of a project. The eighth chapter discusses project issues, in particular project management and project organization. Special consideration is given to the fact that many projects are just one out of many in a project portfolio, and that they may be performed by globally distributed teams.

Up-to-date tools for professional information systems development today are presented and evaluated in the final chapter. This includes tools that support the work of distributed project teams which have team members collaborating on several projects at the same time.

Before I started to work on this book, it seemed to be a clearly defined and overseeable project. However, as the work progressed, practically all topics revealed an abundance of facets asking for investigation. An empirical observation made in many ISD projects came true in this book project as well: "It takes longer than expected and costs more than expected."

I would not have been able to complete this book within a finite time without many people helping me. With sincere apologies to those whom I might have forgotten, my special thanks go to the following people:

Anna Jankowska developed a template for the layout and formatting of the manuscript and wrote many intricate Word macros to make it look like what the reader sees today.

Elvira Fleischer spent months of her life creating figures for the book and fighting with the template while formatting most of the chapters.

Brandon Ulrich deserves special thanks for many roles, including his roles as a contributor of essential ideas covered in the book, author of chapter 9, reviewer of the manuscript and proofreader as well.

Francesca Olivadoti was a valuable help in improving my English writing. Being actually a student of political science in the UK, she may have turned into an IS specialist after proofreading the manuscript twice.

Ilja Krybus helped me with various aspects of process models, modeling with UML and several tools to create reasonable examples (and figures) in chapters 4, 5 and 6.

Armin Boehm created among other things Visio diagrams and screenshots to illustrate standard software implementation and project management.

Ivo Stankov contributed his creativity in formatting several chapters using Anna's template and editing figures for the final layout.

Finally I would like to thank my wife *Kirti Singh-Kurbel,* not only for her patience and the time I could not spend with her during the past two years, but also because she contributed practical insights and experiences from large-scale software development, including project organization, project management and ISD offshoring, in many discussions at home.

Berlin, April 2008 Karl Kurbel

Table of Contents

1 The Digital Firm

The basic question we will answer in this book is: How can an organization today obtain the information systems it needs? What does it take to ensure that those systems are of a good quality and that they work together properly, supporting the needs of the organization?

Focus on business firms

The type of organization we have in mind is a business firm. However, the fundamental principles, methods and technologies for creating information systems discussed in this book are applicable to other organizations such as nonprofit organizations, government offices, and municipal authorities.

Initially, information systems development was mainly technical. It has since evolved into an activity with strong management involvement. Managerial-level decisions are required throughout the entire process. One reason for this is that many different ways to obtain an information system exist today. Managers have to decide which one to follow. For example, an organization may choose to:

Management decisions

1. develop the system inhouse if it has an IT department with a software development group,
2. contract an external partner, a software firm, to develop the system,
3. buy or license standard software and implement it within the organization, provided that standard software matching the firm's requirements is available on the market,
4. buy or license standard software if it satisfies at least some essential requirements, and extend that software with internally-developed components,
5. search the open-source market for complete information systems or for suitable components, and adapt that software to the needs of the organization,
6. search for web services available on Internet servers that would fulfill the desired tasks, and embed those services as part of a an overall solution tailored to the needs of the organization.

Software may be standardized, but organizations are not

Many more variations and diversifications of these approaches are possible, as are combinations of these approaches. One observation from the real world is that standard software rarely addresses the exact information needs of a particular organization. While the software is standardized, organizations are not. This is why standardized software must usually be customized to the organization – entailing minor or major changes to the software. Some important functionalities may be missing, while other features provided by the standard-software developers are superfluous to the implementing organization's needs.

Customization

Adapting standard software to the requirements of an individual organization is called *customization*. Customizing standard software has become a common approach to obtaining individual information systems in most companies today. Different approaches for customization are in use, e.g. parameterization and APIs (application programming interfaces). These approaches will be discussed in chapter 7.

Outsourcing

An even more fundamental management decision with long-term consequences is to entirely or partially *outsource* information systems development. This is a strategic decision because it influences the organization's future options on how to obtain new information systems and run them. *Outsourcing* means to contract out business functions or business processes to a different organization – in the context of this book usually to a software firm – or to a subsidiary.

Offshoring

When the outsourcing partner is located in a different country or continent, then this type of outsourcing is called *offshoring*. Transferring work to low-wage countries in general, and in the IT (information technology) field in particular, has recently received substantial attention. Many organizations hope to benefit from the global distribution of work by offshoring because it cuts costs. India, China and Eastern Europe are the preferred locations for offshoring IT work today. In chapter 2, outsourcing and offshoring with regard to information systems development are discussed in more detail.

1.1 The Role of Information Systems

Businesses rely on information systems

Information systems are the foundation of doing business today. Most business firms would not be able to operate without their information systems. In a similar way, nonprofit organizations, educational institutions, governments, public administrations and many other entities also rely on information systems.

The term *information system (IS)* derives from the fact that such a system deals with information – processing and producing information and making it available to people or other information systems that need the information to do their work. The information systems discussed in this book are used within organizations to support human task solving, automating some of this work where possible. In business informatics, information systems are often defined as socio-technical systems, or as "man – machine – task" systems. These terms indicate that an IS is a technical solution to a task in which human beings in an organization are involved, using the information produced, or providing information to be processed by the system.

Many definitions of the term "information system"exist

Definitions of the term "information system" vary. Depending on the backgrounds and viewpoints of the authors, some focus more on the technical perspective, others on the organizational and management aspects. In the field of management information systems (MIS), for example, an information system has been defined as a set of interrelated components that collect (or retrieve), process, store and distribute information to support decision making and control in an organization [Laudon 2007, p. 14].

In order to balance technical, organizational and management perspectives of IS, we give the following definition:

Definition: information system

> An *information system (IS)* is a computer-based system that processes inputted information or data, stores information, retrieves information, and produces new information to solve some task automatically or to support human beings in the operation, control and decision making of an organization.

The notion of a system implies that there are interrelated elements. In an information system, these elements may be programs or program modules, databases, data structures, classes, objects, user-interface forms or similar entities, depending on the perspective and on the abstraction level of the viewer. Taking a broader view, organizational units and hardware components may be included as well.

IS are composed of software elements and operate within organizations

In this book, we will consider information systems primarily as systems composed of software elements that are developed by and operate within organizations. The modeling and development of information systems, for example, will be discussed from the viewpoint that it is people who develop the software and use abstract models to do so. Therefore the final outcome is software that will be used by people in an organization.

Narrowing the perspective to some extent, we can say that an information system is a software system. Sometimes the two terms will be used interchangeably in this book. However, not every software system qualifies as an information system. Purely technical systems that do not have any organizational impact – for example software switching data packets in a GPRS (general packet radio service) network, a compiler or a cache manager – are not considered information systems even though computer scientists tend to call purely technical systems such as the last one "managers" or "management" systems.

Information systems are playing an increasingly important role in most organizations today. In many industries, companies depend heavily on their information systems. Information-intense industries such as insurance, banking and telecommunications could not survive without information systems. Some industries would not exist without IS, and electronic commerce would not have been invented. Firms such as Amazon, Yahoo, Travelocity, Hotels.com etc. would simply not have been created without powerful supporting information systems.

All industries depend on information systems

Also in traditional industries such as manufacturing and retail, there is a growing dependence on information systems. Firms need IS for every part of their business – for their daily operations, for controlling and reporting, for their strategic planning, and for maintaining their supplier and customer relationships. It would be inconceiveable that General Motors, Siemens, Wal-Mart, Metro etc. are doing business today without efficient information systems.

Another reason why information systems are so important is that information technology (IT) accounts for a significant share of capital investment in modern economies. In the US, for example, investment in IT has become the largest single component of capital investment – 35 % of private business investment, and more than 50 % in information-intense industries such as the financial sector [Laudon 2007, pp. 5-6].

It is a well-known fact that efficient usage of information technology presupposes information systems able to utilize and exploit the features of the technology. Business productivity can increase substantially and firms can achieve strategic advantage over their competition by deploying information systems that support their strategic goals.

Nicholas Carr: "IT doesn't matter"

However, the importance of information technology as a differentiating factor in organizational performance has been challenged by Nicholas Carr in a famous article entitled "IT doesn't matter" [Carr 2003] and in his subsequent book "Does IT matter? Information technology and the corrosion of competitive advantage" [Carr 2004]. Carr argues that information technology may be bought by any company in the marketplace, so competitive advantage obtained through IT can be easily copied. Therefore, IT has become a commodity rather than a strategic factor.

Carr's theses have stirred-up an intensive discussion in the IS and management communities. Most IS experts disagreed with Carr's theses, yet one effect of the discussion was the significantly increased pressure on IT departments to justify the return on information-technology investments.

"Does software matter?"

With the question: "Does software matter?" Carr continued his argument and also classified software as a commodity that will mostly be developed in software factories in low-wage countries, bought off the shelf, or obtained as a service on a plug-and-pay basis [Carr 2005].

While some of Carr's observations are certainly correct, the situation in typical organizations around the world is more nuanced. Complete off-the-shelf software packages are suitable for standardizable products such as office programs but not for the heavyweight enterprise systems managing the business processes of a firm. Even if some components are purchased as ready-to-install modules or developed in India or Bangladesh, they still need to be integrated into and adjusted to the diversified information systems landscape in the organization.

Developing IS is not the core business

The situation is as outlined in the beginning of this chapter: Some IS may be developed inhouse, some bought off the shelf, and others purchased and customized. We definitely agree with one point of Carr's arguments: Typical business firms whose core business is not software have reduced the volume of internally developed corporate software dramatically, sometimes to the extent that they do not develop new software at all any more. The role of information systems development has changed – from developing entire new business solutions inhouse to implementing what others have developed, integrating that with the rest of the information systems in the organization, and perhaps developing some supplementary components.

1.2 Information Systems in the Enterprise

Stand-alone information systems

In the early times of business computing, most information systems were designed to solve specific problems or support a particular function, such as MRP (materials requirements planning), payroll or financial accounting. These were stand-alone systems, developed only to solve or support the task at hand. They were "islands" not connected with one another.

Integrated information systems

A typical enterprise today uses a large number of information systems. These systems tend to be integrated so that they can work together. All major business processes are represented in and operated with the help of information systems. Fewer and fewer companies use systems that they developed themselves. Instead they work with standard software, customized and extended to their needs.

Standard software, application package

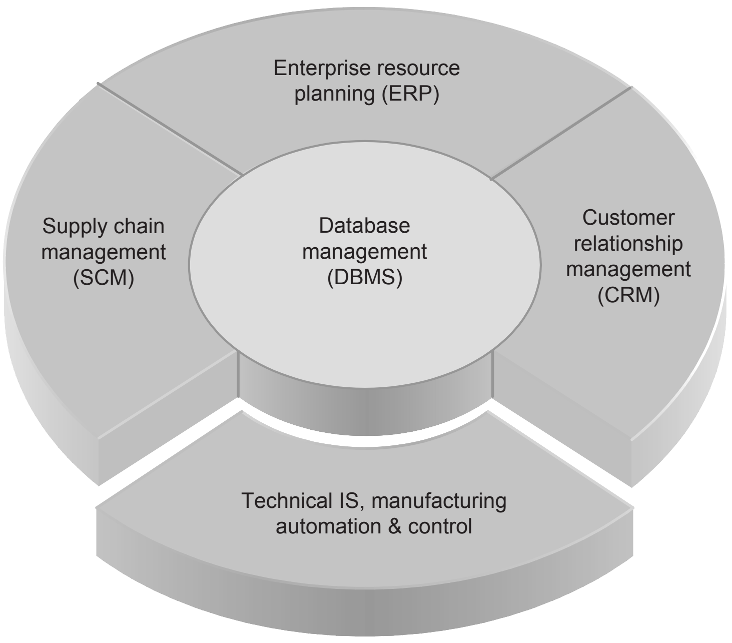
With *standard software*, also called standard packages, we denote a software system that was developed with the aim of being used by many organizations. Standard software exists for many problem areas: office programs, database management systems, enterprise resource planning etc. When business problems are underlying the software, the terms *business software* or *application package* are sometimes used.

Core information systems: ERP, SCM, CRM

A typical configuration of information systems in an enterprise comprises at least three large systems as figure 1-1 illustrates: An ERP (enterprise resource planning) system, an SCM (supply chain management) system and a CRM (customer relationship management) system.

All are built on top of one or more database management systems (DBMS) – ideally using the same logically integrated database.

Figure 1-1 **Core information systems in a typical enterprise**


Enterprise resource planning (ERP)
Supply chain management (SCM)
Database management (DBMS)
Customer relationship management (CRM)
Technical IS, manufacturing automation & control

The ERP, SCM and CRM systems are usually standard software that have been customized according to the requirements of the individual organization. Nowadays these three types of systems tend to be integrated: An SCM module, for example, will have access to information available in the ERP system.

CAD, CAP, CAM, CNC

If the company is a manufacturing firm, then technical information systems and software for manufacturing automation and control will form an equally significant share of the corporate software as the business systems. Ideally, technical systems such as CAD (computer aided design), CAP (computer aided planning), CAM (computer aided manufacturing) and CNC (computerized numerical control) will be well-integrated with the business systems, using the same logical database.

IS on a network

Information systems such as the above are designed to be accessed by many users at the same time. Previously these systems were run on one computer, usually a large mainframe, and users were connected through terminals. Nowadays most processing is distributed to various computers connected by a network. Users access IS functionality as clients from personal computers, workstations, terminals and other end devices over the network. This means that the desired functionality must be available on network servers.

Application servers

A server providing access to information systems functionality is called an *application server*. Before the web age, that term referred to a server in a client-server based system. Nowadays application and web functionalities have become closely related. Therefore application servers and web servers are partly sharing the work, with some overlap, and many application servers are becoming web based. Well-known products include BEA WebLogic, Borland AppServer and IBM WebSphere Application Server. Open-source application servers are Apache Geronimo and JBoss.

IS development means development around the core systems

Organizations use more information systems than those depicted in figure 1. Dedicated systems for particular problem areas can be found in vast numbers. Yet the ones contained in the figure may be regarded as the core information systems on which today's companies operate. Information systems development today normally means development around those systems. The core systems are already there, limiting the degree of freedom for new systems or making additional systems unnecessary because the functionality is available in the standard software. What can be done and what has to be done is often determined or constrained by the requirements of the core systems. Any additional system must collaborate with the existing ones, in many cases providing data as input or processing information produced as output by the core systems.

1.2.1 Enterprise Resource Planning

An ERP system is an organization's IS backbone

The most fundamental information system in most organizations is the enterprise resource planning (ERP) system. An ERP system is a comprehensive information system that collects, processes and provides information about all parts of an enterprise, partly or completely automating business processes and business rules within and across business functions [Kurbel 2005]. ERP systems cover all major business functions and processes. They have reached a high degree of maturity because they have been around for many years. ERP systems often originated from former MRP II (manufacturing resource planning) and MRP (material requirements planning) systems that go back as far as the late 1960s and early 1970s.

Is an ERP system a single system?

ERP systems are very large systems, so the question may arise whether an ERP system is actually one information system or many. ERP systems cover many areas and thus contain many modules. Originally these modules were more or less separate. A synonym for standard software was "modular program" because modules covering certain business functions were only loosely coupled and could be implemented separately. In this case, we might say that each module is an information system of its own.

Nowadays the degree of integration between the modules of an ERP system is so high that the systems appear as one system. For the user, an ideal ERP system will behave like *one* enterprise-wide information system with one database and one common user-interface. Therefore we consider an ERP system as one information system. Nevertheless such a system may be composed of many subsystems and many databases, as long as they are well integrated.

Market leader in ERP

The most common ERP system worldwide is SAP ERP (formerly SAP R/3). Its wide range of functionalities are illustrated in figure 1-2. Five comprehensive areas are covered by SAP ERP:

ERP application domains

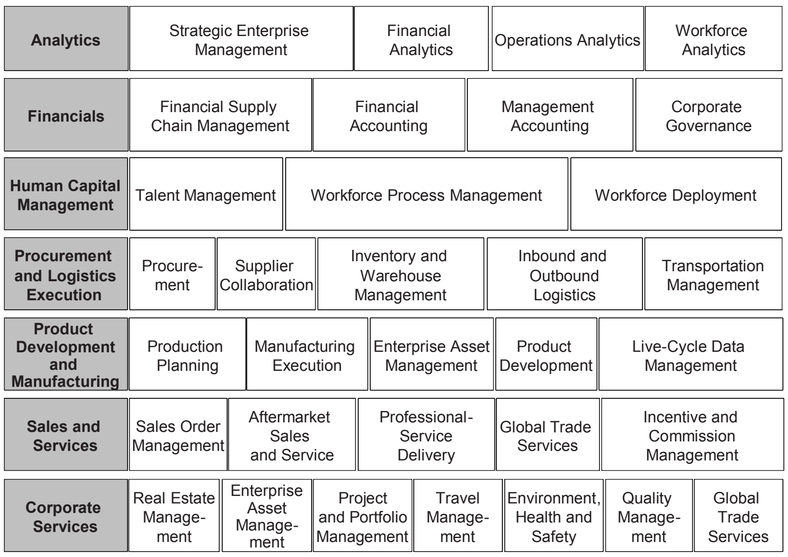
- Analytics (support for strategic enterprise management and for reporting, planning, budgeting, analyzing most other areas)
- Financials (financial and managerial accounting)
- Human capital management (employee management, transactions involving employees, payroll etc.)
- Operations (logistics and production planning and control, inventory and warehouse management, procurement, sales etc.)
- Corporate services (services supporting employees in real estate management, incentive and commission management, travel management and more)

Figure 1-2 shows only the top-level domains supported by SAP ERP. Each section can be decomposed into many further sublevels. At the lowest sublevels, very detailed functions for each step of each business process are provided.

SAP, Oracle, Microsoft and open-source

On the market there are many ERP products offering similar functionalities although they may be arranged in different ways. However, their market shares are rather small. The big players are, after a round of mergers and acquisitions at the beginning of the 21st century, SAP, Oracle (comprising former PeopleSoft and J.D. Edwards) and Microsoft. A number of ERP systems are available as open-source, including Compiere, ERP5, Openbravo ERP and OFBiz [Serrano 2006].

Figure 1-2 **Application domains and modules of SAP ERP [SAP 2007b]**



Analytics	Strategic Enterprise Management	Financial Analytics	Operations Analytics	Workforce Analytics			
Financials	Financial Supply Chain Management	Financial Accounting	Management Accounting	Corporate Governance			
Human Capital Management	Talent Management	Workforce Process Management	Workforce Deployment				
Procurement and Logistics Execution	Procurement	Supplier Collaboration	Inventory and Warehouse Management	Inbound and Outbound Logistics	Transportation Management		
Product Development and Manufacturing	Production Planning	Manufacturing Execution	Enterprise Asset Management	Product Development	Live-Cycle Data Management		
Sales and Services	Sales Order Management	Aftermarket Sales and Service	Professional-Service Delivery	Global Trade Services	Incentive and Commission Management		
Corporate Services	Real Estate Management	Enterprise Asset Management	Project and Portfolio Management	Travel Management	Environment, Health and Safety	Quality Management	Global Trade Services

© SAP AG

Since most systems provide the functionality required for enterprise resource planning, businesses usually do not develop information systems for ERP any more. However, if a company finds that some "standard" solution provided by the chosen ERP system does not reflect its individual requirements appropriately, then that company is likely to look for its own solution. This could be by developing or purchasing a dedicated information system for the specific problem, extending the ERP system, modifying its programs, or in other ways working its way "around" the ERP system.

Restrictions set by the ERP system

The new solution has to meet technological restrictions that are set by the ERP system. These restrictions could be the platform on which it runs, the programming language (if program code has to be modified), the database management system etc.

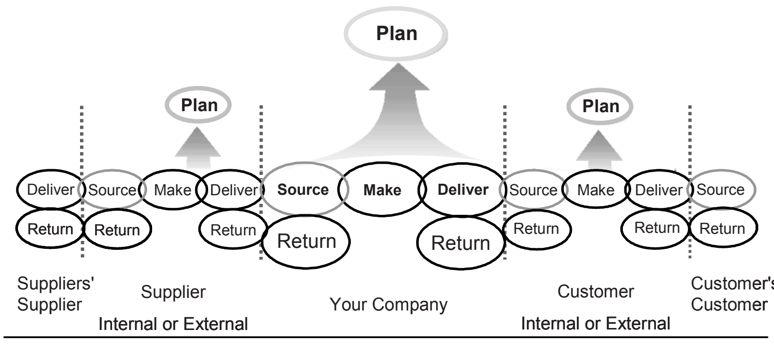
While the focus of ERP is to support internal business processes, business activities do not end at the boundaries of the company. Going beyond these limits is the task of supply chain management (SCM).

1.2.2 Supply Chain Management

Organizations collaborate in supply chains

The second of the core information systems is an SCM system. Organizations are collaborating in supply chains, creating networks of suppliers and customers over many levels, including the suppliers' suppliers and the customers' customers, as shown in figure 1-3.

Figure 1-3 **Supply-chain processes [SCC 2006]**



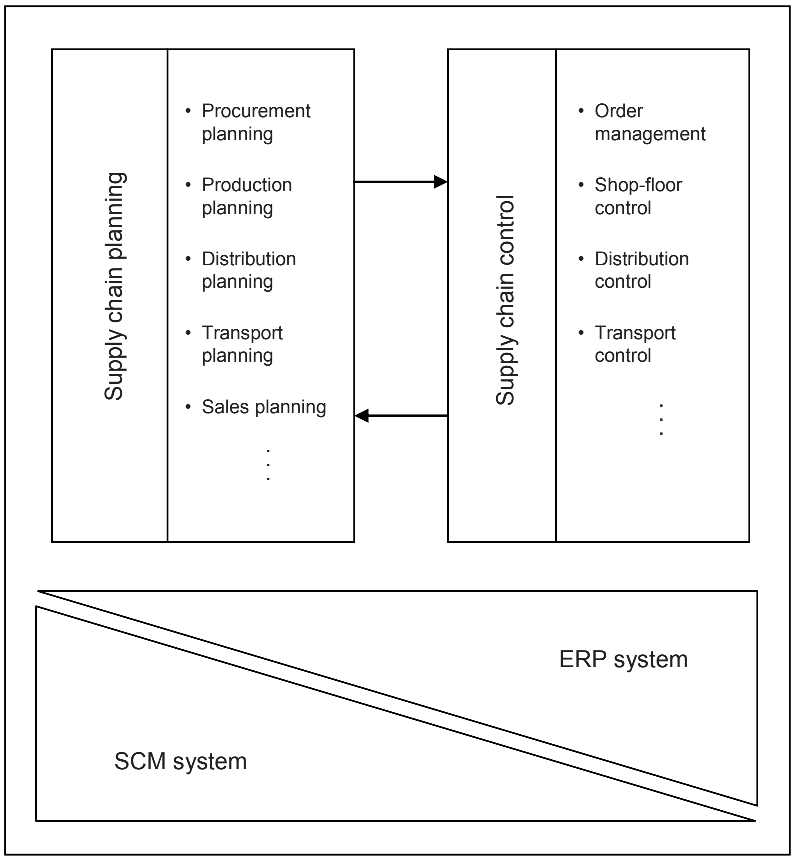
A firm's performance depends on its supply chains

Businesses have become increasingly specialized. They concentrate on their core competencies, outsource secondary activities and purchase assemblies rather than manufacture them themselves. Consequently, effective supplier-customer networks have become crucial for success. The performance of a firm depends heavily on the smooth functioning of the supply chains to which it belongs. No matter how efficient internal processes and the supporting ERP system are, if the supplier of a critical component, or that supplier's supplier, or a supplier further up in the chain fails to deliver properly, the company will not be able to perform as it thought it could. This effect is illustrated by figure 1-3.

SCM looks at the business partners

Therefore, a natural extension of ERP is *supply chain management* [Ayers 2001]. SCM considers the organization's business partners, in particular the suppliers and their suppliers. In addition, many methodological and technical shortcomings of ERP have been removed or at least improved in SCM. These improvements are known as *APS (advanced planning and scheduling)* [Meyr 2002] and are implemented in SCM solutions by SCM vendors.

Figure 1-4 **Relationship between SCM and ERP [Corsten 2001]**



SCM and ERP are connected

Supply chain management and enterprise resource planning are closely connected. This is due to two facts: In a supplier-customer network, many results provided by ERP are needed as input for the SCM system and vice versa. Secondly, the same functions are sometimes needed in both systems. There is a natural overlap between ERP and SCM functionality. In closely coupled solutions (e.g. SAP SCM [SAP 2005b] and SAP ERP), the SCM system may even invoke functions of the ERP system.

Planning and control levels in SCM

Like ERP systems, SCM systems support all levels of planning and control, from long-term strategic planning (such as setting up a supplier-customer network) to execution of daily operations. Figure 1-4 shows the relationship between ERP and SCM systems on the mid-range planning and control level. Dedicated planning functions are found in the SCM system, whereas control functions are often the same as in the ERP system. In addition, there is close interaction between the two systems because they often use the same data.

1.2.3 Customer Relationship Management

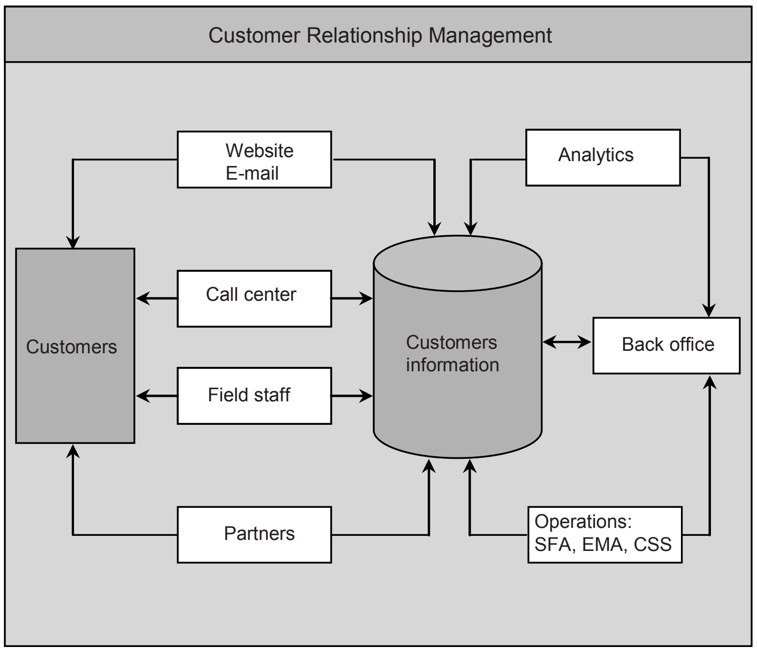
CRM: identifying, acquiring and retaining customers

The most recent member of a typical business information systems suite is a customer relationship management (CRM) system. CRM is an integrated approach to identifying, acquiring and retaining customers [Siebel 2006]. Some authors consider good customer relations the most valuable asset of a business firm. While marketing and management have always placed high importance on customer relationships, the business's information systems have not supported this view until the late 1990s. Previously, valuable customer information was distributed and maintained in various information systems – in the ERP system, in e-commerce, call-center, customer-service systems, and more.

Managing customer interactions

The need to place the focus on customer relationships arose when marketing, sales and service departments developed new channels beyond traditional ones such as retail stores and field sales: websites (electronic shops), e-mail ordering, call centers, mobile commerce, push services etc. The number of sources of customer information grew. It became increasingly difficult to find, maintain and update customer information efficiently and consistently. Analyzing customer data for marketing in a unified way, in order to generate more value for the firm, was not possible. By enabling organizations to manage and coordinate customer interactions across multiple channels, departments, lines of business and geographical regions, CRM helps organizations increase the value of every customer interaction and improve corporate performance.

Figure 1-5 **Sources and uses of customer information [Siebel 2006]**



Definition of a CRM system

A CRM system is an information system that is used to plan, schedule and control the presales and postsales activities in an organization [Finnegan 2007, p. 4]. The goal of CRM is to improve long-term growth and profitability through a better understanding of customer behavior. CRM includes all aspects of dealing with current and prospective customers: call center, sales force, marketing, technical support, field service etc. All customer information from these sources is collected and maintained in a central database as illustrated in figure 1-5. Marketing, sales and service departments access the same information.

"Back office" and "front office" systems

A typical "back office" system the CRM system is connected to is the company's ERP system. CRM systems are sometimes called "front office" systems because they are the interface with the customer.

CRM systems are composed of operational and analytical parts. *Operational CRM* includes in the first place support for:

Operational CRM: SFA, EMA, CSS

– SFA (sales force automation – e.g. contact/prospect information, product configuration, sales quotes, sales forecasting etc.)
– EMA (enterprise marketing automation – e.g. capturing prospect and customer data, qualifying leads for targeted marketing, scheduling and tracking direct-marketing etc.)
– CSS (customer service and support – e.g. call centers, help desks, customer support staff; web-based self-service capabilities etc.).

Analytical CRM

Analytical CRM consolidates the data from operational CRM and uses analytical techniques to examine customer behavior, identify buying patterns, create segments for targeted marketing, identify opportunities for cross-selling, up-selling and bundling, and separate profitable and unprofitable customers. This is done with business intelligence techniques such as OLAP (online analytical processing) and data mining, based on a data warehouse.

ERM, PDM

In addition to operational and analytical customer relationship management, many CRM systems include components for ERM (employee relationship management) and PDM (partner relationship management). This is due to the fact that employee performance and partner (e.g. dealer) performance are closely related with customer relationships.

Connections between CRM and various parts of enterprise resource planning are quite tight. That is why ERP vendors also provide CRM systems which interoperate with their respective ERP systems. It is not surprising that the long-time market leader in CRM, Siebel Systems [Siebel 2006], was bought by Oracle in 2006.

1.2.4 Database Management

All of the above information systems handle large amounts of data. Only in the early days of business information processing were these data stored in program-related data files. Early MRP (material requirements planning) systems, for example, had quite sophisticated file organization systems. However, today all non-trivial business information systems store their data in databases.

The roots of database management systems go back to the 1960s and 1970s, so it is not surprising that today's systems have reached a high level of maturity. The functionality of a modern DBMS comprises a lot more than just storing and retrieving data. For example, database schemata can be generated automatically from models. Visual tools for semantic data modeling, for creating graphical user-interfaces and for querying the database, as well as a workflow management system and much more are provided. In fact, Oracle's entire ERP functionality is largely based on tools around Oracle's database management system. This is not surprising as Oracle Corp. is one of the world's largest DBMS vendors.

Definition: database management system

A DBMS is an information system that handles the organization, storage, retrieval, security and integrity of data in a database. It accepts requests from programs or from end-users, processes these requests and returns a response, e.g. transferring the requested data.

Most DBMSs are relational

Most of today's database management systems are relational systems (RDBMS). With the emergence of object-oriented analysis, design and programming, RDBMS were extended to accommodate not only data records but also objects, thus realizing object persistence. Notwithstanding the existence of dedicated object-oriented DBMS, the majority of business information systems use RDBMS.

Database management systems on the market

There are many relational database management systems on the market. Oracle (Oracle Database), IBM (DB2), Microsoft (SQL Server) and Sybase (Adaptive Server Enterprise) have the largest market shares. MySQL and PostgreSQL are popular open-source products. A widely used DBMS for end-users, but not for large professional business systems, is Microsoft Access.

A major achievement of more than four decades of business information processing was the decoupling of application systems and database management systems. In earlier times the programs of an MRP II or ERP system, for example, referenced the DBMS directly. Since each vendor's DBMS implementation had its own extensions and modifications of the SQL (structured query language) standard, the application system and the database management system were tightly coupled. Portability of a database – and thus of an entire ERP system, for example – was a difficult, sometimes impossible task.

Interfacing with an RDBMS

Nowadays an RDBMS supports common interfaces with standard access methods. Programs now invoke operations provided by the interfacing technology instead of directly accessing the database management system. Portability has significantly improved in this way. Standard technologies and access methods are:

– ODBC (open database connectivity), providing access to databases on a network for Windows programs,
– JDBC (Java database connectivity), allowing Java programs to access a relational database via the SQL language,
– Java EE/EJB (Java enterprise edition/Enterprise JavaBeans), giving higher-level access to a database than JDBC, using EJB entity beans,
– XML (eXtensible markup language) enabling, providing standard access methods for navigation and queries in XML. Data are extracted from a database and put into XML documents and vice versa.

The functionality of a professional DBMS is provided on a server. Like an application server for the business functionality, a database server is connected to a network. ERP, SCM and CRM functions access the server over the network. Human users such as database administrators and end-users also reach the server over the network.

1.2.5 Electronic Commerce and Electronic Business

With the explosive growth of the Internet, organizations began to employ the web to do business. Many organizations developed web-based systems to present themselves and to advertise and sell their products.

Web technology is different from conventional IS technology

This posed a major problem since web technology is quite different from the conventional information systems technology the back-office systems are based on. Web-based systems are written in HTML (hypertext markup language) and in software technologies extending HTML, whereas a typical ERP system is written in a language such as Java, C, C++, Cobol etc. and strongly relies on a database management system.

Two lines of development emerged: 1) dedicated web-based information systems and 2) web-based front-ends for the core back-office systems. In the beginning, web-based systems were stand-alone systems, not integrated with the business processes and the ERP/CRM systems of the company. This was not only a technological problem but also an organizational one.

Electronic commerce

Electronic commerce (e-commerce) refers to the process of buying and selling products or services over a digital network. Usually it is assumed that this network is the Internet and that the products or services are offered via the World Wide Web. An *electronic shop* (or a web shop) is an information system that presents products and services in a product catalog. It lets customers add products to a shopping cart and complete the purchase with a financial transaction. Product configuration, personalization and many more features may be included.

Data redundancy between ERP and e-commerce systems

A fundamental problem in the development of an electronic shop is that most of the data involved are available in the company's database or have to be stored in that database. Therefore, the shop system needs to access the database. Technologies to access a database from an HTML based user interface are available, for example invoking stored procedures of the database from ASP (Active Server Pages) or JSP (JavaServer Pages) scripts. Yet the script code is likely to contain redundant data-related functions that are implemented in the ERP system anyway. If the ERP system and the electronic shop are not integrated, this redundancy cannot be avoided. Many more problems may arise from the lack of integration.

Until today, e-commerce systems were often developed as individual solutions, without employing standard software. Ready-made shop solutions with tools for adaptation to company specific features are available, yet many organizations prefer tailor-made systems developed inhouse or by a web design agency.

Electronic business

Electronic business (e-business) takes the concepts and technologies of e-commerce into the inside of the business firm and into the business relations with partners. E-business is business performed with the help of digital networks, based on Internet, intranet, and web technology. E-business comprises all the business processes in the company, including processes for the internal management of the firm and for coordination with suppliers, customers and other business partners. E-commerce is a part of e-business.

One of the implications of e-business for information systems and their relationships is that system communication and interaction with users are now increasingly based on Internet protocols and languages instead of proprietary communication mechanisms. For example, a typical graphical user interface (GUI) of an ERP system in the past was based on forms that were generated with a tool provided by the ERP vendor. Using web technology in e-business now means that the user interface will not be created in a proprietary GUI technology but written in HTML or created with a tool that generates HTML forms. Likewise, data communication between systems or system modules is moving to

Internet technologies. Data are increasingly transferred in XML format, not only between web-based systems but also for accessing databases.

Coupling ERP, SCM, CRM with an enterprise portal

Another implication is that organizations provide portals for their employees, for business partners such as customers and suppliers, and for the general public. An *enterprise portal* is a website that serves as a single entry point or gateway to a company's information and knowledge base for employees and possibly for customers, suppliers, partners, or the general public. In modern architectures, access to the functionality and data resources of the core business information systems is also provided through portals. This means that systems for ERP, SCM, CRM etc. have to be coupled with a portal – another challenge where web technology and conventional software technology meet.

Web server and application server

Since electronic commerce and electronic business usually employ web technology, the basic pattern of client requests and server responses applies. This means that e-commerce/e-business information systems need a *web server*. If they are integrated with the core information systems that run on an application server, then both a web server and an application server will be present. The two servers communicate with respect to application functions and data. Since the functionalities of web and application servers are overlapping, a division of labor between the two has to be established.

E-commerce and e-business started as approaches employing cable-based networks and desktop computers. With the emergence of wireless networks and end devices capable of receiving, displaying, and transmitting data at reasonable speeds – mobile phones, PDAs (personal digital assistants), pocket PCs – a performance similar to that available on stationary computers was desired for mobile workers and their mobile devices.

Mobile commerce, mobile business

Mobile commerce (m-commerce) and *mobile business (m-business)* are the counterparts of e-commerce and e-business when the respective activities are based on the use of mobile appliances and wireless network technologies. Such technologies are, for example, UMTS (universal mobile telecommunication system), i-mode (an NTT DoCoMo technology [NTT 2006]), GPRS (general packet radio service), HSCSD (high speed circuit switched data) and GSM (global system for mobile communication).

Implementations of mobile-commerce and mobile-business systems vary significantly, depending on the type of network, the protocols available on the mobile devices, and the computing power of the devices. While early mobile phones were more or less "dumb" terminals, just capable of displaying simple data on WML (wireless markup language) cards, many modern phones have XHTML MP (eXtensible HTML mobile profile) browsers or are Java enabled. This means, for example, that they can serve as "fat" clients and execute Java programs themselves.

1.3 The Role of Information Systems Development

Summing up the discussion in the previous sections, the environments of business information systems are quite diversified. We start the examination of the role of IS development today with a discussion of the technological infrastructure of information systems.

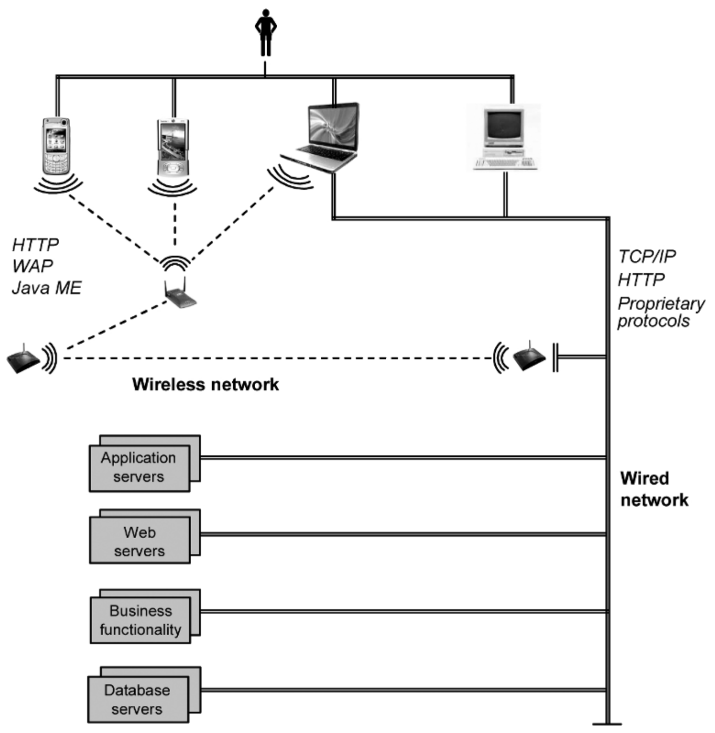
1.3.1 Technological Infrastructure of Information Systems

Typical features that many organizations share are summarized in figure 1-6. While the back-ends are more or less similar, the front-ends differ substantially. The core information systems of an enterprise are usually built on top of a database management system. DBMS functionality is available on servers on a network. The business functionality of a firm's information systems is provided by application servers that are also accessible over the network. If the network is the Internet and web technology is used, then web servers talking to the application servers have to be integrated.

Network protocols

Users access information systems from end devices, typically over a network. If the network is a stationary one, then TCP/IP (transmission control protocol/Internet protocol) and HTTP (hypertext transfer protocol) for Internet/intranet and web based systems, or proprietary protocols for conventional systems are used. In the wireless networks mentioned above, data are transported via HTTP, WAP (wireless access protocol) or Java ME (Java mobile edition) technology between the end-user's device and the web server.

Figure 1-6 **Technological infrastructure of information systems**



1.3.2 What Happened to ISD?

"Does ISD matter?"

After the previous discussion, one might be tempted to ask: What is left of information systems development (ISD) if "everything is there?" Information systems development used to be a classical discipline and an integral part of business informatics, computer science and information systems programs. Doesn't ISD matter any more?

The "nothing is there" hypothesis

The answer is that the focus of ISD has shifted. In the past, the study of and approaches to the development of information systems started from the assumption that "nothing is there." Or more precisely, the basic assumption was that the organization either did not have an IT based solution, or that it had an old information system and wanted to replace it with a new one. In the first case, the organization would start its development efforts from scratch; in the latter case, it would develop a new and better system based on an analysis of the old one.

Is everything there?

The "everything is there" view in fact needs a closer look. First, where does "everything" come from? Of course, there are professional organizations that still develop large-scale standard software such as an enterprise resource planning system, a database management system and others.

Second, not really "everything" is there. There are gaps in the standard solutions provided by the vendors of application packages. The gaps have to be filled by individual information systems. Likewise, additional IS are needed when new requirements arise. For example, organizational requirements may change with business strategies, marketing needs, emerging new technologies etc.

1.3.3 Scenarios for Information Systems Development Today

Obviously the situation regarding information systems development is different in organizations whose core business *is not* software, and in those whose core business *is* software.

User organizations

One characteristic of the first category (user organizations) is that the development of new information systems has significantly decreased. Nowadays, organizations develop fewer systems on their own, or none at all any more. Instead, they employ standard software and technologies as summarized in figures 1-1 and 1-6, and adapt and extend the standard software. New, more powerful, and perhaps better versions of the software can be obtained when the vendor provides a new release, and if the organization decides to buy and implement that release.

More precisely, five idealistic scenarios for ISD can be distinguished.

Scenario 1: Patchwork and niches

Customizing

An organization licenses and implements standard software and customizes that software with the help of the customization tools available for the software, e.g. parameterization or model-based generation. Missing features are added via APIs (application programming interfaces). This means that some parts of the overall solution are developed specifically for the individual organization, either inhouse, by a software firm, or by the vendor of the standard software.

Extending standard software

If an entire problem area relevant to the organization is not covered by the standard software, then a complete information system will be developed either inhouse or by a partner (see scenario 4), or purchased from a different vendor. This new system has to fit the rest of the company's information systems, not only regarding technology but also integration on a logical level. Restrictions and requirements for the new IS are set by the organization's information systems architecture. If the new system does not match those restrictions and requirements, bridges have to be built to make the new and the existing systems compatible ("bridge programming"). Connecting software products from different sources or vendors can be a non-trivial development problem.

New releases of the standard software may create problems for add-on systems so that those systems need to be modified.

Scenario 2: Personal information management

Definitely not all tasks at all individual workplaces in an organization are supported by standard business information systems, yet most people today use a personal computer for their daily work. Almost all white-collar workers do, as do many blue-collar workers as well. Many workplace related tasks, on the personal level, are solved with the help of office programs.

Development by end-users through visual tools and IDEs

While simple problems may be addressed using those tools directly, more complicated tasks require the development of programs (often called "macros") or entire information systems. With end-user oriented tools and languages, workers can develop themselves, or have someone develop for them, quite powerful information systems based on Microsoft Excel and Access, for example. Excel and Access support end-user development through visual tools and the VBA (Visual Basic for Applications) language. As the level of IT education in the business world is increasing, adept professionals may also develop larger solutions for their individual tasks in a convenient IDE (integrated development environment) such as Visual Studio .NET.

A typical example is data analysis

Typical examples of personal information management are end-user systems for *data analysis*. Using data provided by one of the standard business systems, an information system in Excel based on pivot tables may be employed at the workplace to analyze the data and create nice-looking charts. Many ERP, SCM, and CRM systems today provide download and upload features and interfaces for office programs such as Excel, Access, and Outlook. SAP and Microsoft even developed a common software (called Duet; http://www.duet.com) to enhance integration of SAP application software (such as SAP ERP) with personal information management (based on MS Office Professional).

Software organizations

By *software organizations* we denote professional software firms that live off producing and selling software, as well as IT departments, software development groups and subsidiaries of large organizations whose primary task is to produce and maintain software for their parent organizations. A special type of software organization, with blurry edges, are loose networks of developers that create open-source software.

Scenario 3: Large-scale development

Entire information systems such as ERP, SCM or CRM systems are usually created by organizations that have the financial power to invest large amounts of money in standard software development and receive the returns only after some years.

Development of standard software rarely starts from scratch

Although this type of development is not constrained by an existing inhouse IS landscape like a user organization has, development rarely starts from "nothing is there" either. Unless the problem domain is a completely new one, some older system or at least some modules are likely to exist already. If an ERP vendor, for example, decides to set up a new ERP system, then that company probably has experience in the field from selling such a system. Since not all parts of the old system will be obsolete – a bill-of-materials processor, for example, will always process bills of materials in the same way – some of the old program code is likely to survive into the new system.

Existing systems limit the degree of freedom

Upgrading an existing system is more common than developing an entirely new system from scratch. New versions or releases are produced based on the existing system, adding new features and revising and improving old features. This limits the degree of freedom rather heavily.

The new release has to run in the user organization's social and technological environments, often in identical environments as the old releases.

New systems and releases are often subject to the constraint that interfaces for industry-standard systems in adjacent areas have to be provided. An ERP system, for example, needs interfaces for Siebel CRM, and any non-ERP system will require interfaces for SAP ERP.

Scenario 4: Custom information systems

Although really large information systems are usually standard software (scenario 3), this software may need to be substantially extended for an individual organization. Likewise, entirely new, individual information systems may be needed to fill gaps not covered by the standard software. These types of development are often contracted to professional software organizations. That organization will carry out the development, working together with employees of the user organization at various levels and stages.

Software company and user organization working together

If the user organization has their own IT staff, a typical division of labor is that these people do the requirements engineering and produce a requirements specification document (cf. section 2.2.1). The software vendor will design the system, develop it and deliver it to the customer. There it will be tested and evaluated by business and IT staff.

Scenario 5: Open-source development

Open-source software (OSS) is software that is available as source code to the public free of charge. All types of software exist as open-source: operating systems, database management systems, web servers, office programs, and even business information systems such as ERP and CRM systems.

Sources of OSS systems

The development of OSS comes in many variations. OSS is typically developed around a nucleus – a software system – that was initially created by an organization or individual and then made available to whoever is interested in the code. Many developers around the world revise the code and contribute additional code. Some OS systems started out from hobbyist programming by individuals who wanted to do good to the world (or perhaps bad to capitalist organizations exploiting the world through costly software). Other OSS was initially created by professional organizations and then made available to the rest of the world. The primary reason for doing so is normally not altruism but to earn money from services and software based on the OSS.

Open-source development is incremental and iterative

Open-source development goes on in an incremental and iterative way. Since many people or organizations are involved, revising and contributing code, there are usually rules of how new versions of the software become public – sometimes strict rules, sometimes loose rules, sometimes practically no rules. Nevertheless, anyone interested in OSS may download the code, use it, incorporate it in their own systems, and build new systems based on that code. Legal obligations may have to be fulfilled when revenue is earned from the new systems (see section 4.6).

Outlook

All five scenarios are simplified abstractions of real-world situations, yet they are useful to distinguish different types of information systems development.

Scenario 4 is the one that comes closest to what was underlying classical ISD. Scenario 2 will not be covered in this book. Scenario 1 is the dominating scenario for most user organizations today. Many of the methods and tools discussed in this book apply to large-scale development by software organizations (scenarios 3 and 4) as well as to development around standard software (scenario 1).

While pure OSS projects (developing open-source software within the OSS community) are not the focus of this book, the use of OSS systems or modules within large-scale professional systems (scenario 3) and within custom information systems (scenario 1) is an increasingly important aspect.

2 Managing the Making of IS

The making of an information system is subject to management decisions. Managers are involved in different stages and at different times, making decisions before and during the project. Speaking of "management" in the context of making information systems, at least two levels of management decisions have to be distinguished: senior management and operational management decisions.

Senior management decisions

Senior managers decide whether an information systems project should be started or not. They set the overall framework for the project in terms of budget, resource allocation, staffing, time limits etc.

Examples of management decisions

Management decisions may also need to be made in the course of the project. Should more resources be allocated if the project is late or if the results so far are not good enough? Should we continue with the project or cancel it before more money goes down the drain? Decisions like these have to be based on examinable intermediate results and points in time, so-called *milestones*.

High project-failure risk

Starting, continuing and cancelling a project are highly critical management decisions since the failure risk is rather high. According to industry surveys, only about 30 % of all application software development projects are considered successful [Standish 2004]. Close to 20 % are failures, i.e. they are cancelled prior to completion or completed but never used. The remaining 50 % are challenged – not delivered on time, with cost overruns, lacking features, or not meeting expectations. Notably almost half of the challenged projects exceed their budgets.

Operational management decisions

Project management

Operational managers, in particular *project managers*, are the ones who run a project once it has been decided. Their tasks include project planning, allocating and assigning personnel, scheduling activities using appropriate techniques (e.g. network planning techniques, Gantt charts), controlling costs and time, and more. Project managers also need milestones to help them do their jobs efficiently, e.g. to control time and costs. Usually, their milestones are fine-grained whereas for senior management, milestones are on a higher aggregation level.

In this chapter we will focus on decisions that involve the senior management of a firm although there is certainly some overlap with the tasks of operational managers. (Project management will be discussed in a separate chapter later in this book.)

2.1 Creating the Idea

(Better) solutions desired

Where does the initiative for a new information system come from? A generic textbook answer to that question is: Someone detected a problem or an unsatisfactory situation and is looking for a (better) solution which can be supported or provided by an information system. Examples of such problems or situations are:

- The marketing department feels that the company is reacting far too slowly to changing customer demands, mainly because it does not receive consolidated sales figures by products and sales regions in real-time. Successful competitors perform better because they apparently have efficient customer response systems that provide such information.
- A business process in the organization is not flowing smoothly. The problem is not the business process because that was just recently reengineered. The problem seems to be that the existing information systems do not support the new workflows smoothly.
- New strategic opportunities open up with the emergence of new technologies. For example, to be able to sell products over the Internet requires significant extensions of the conventional information systems landscape.
- Target groups demand new product or service features. Imagine the rise of a new fad: Young career-orientated business professionals will suddenly only wear tailor-made clothes, yet bought off the shelf. Fashion stores will immediately need powerful "configurators", translating measurements automatically into production orders and NC (numerical control) programs, and powerful logistics systems that deliver the clothes within an hour or two to the shop.
- Business software vendors create, or jump onto a new trend, promising significant benefits to their customers and offering solutions for the underlying problems. Companies fearing to miss the train join-in.
- Industry associations address problems and propose new approaches, solutions or information systems to their members. This is often the case with small and medium-size enterprises that do not have the manpower and/or knowhow to observe the information systems market and technological trends.

New technologies trigger new IS

Another starting point for partly new or entirely new information systems is when *new technologies* for software or hardware are introduced to the market. Gradually the hardware or software vendors the firm is depending on will increase pressure upon them to migrate to the new technology, because eventually they will not support the old platforms any more.

For the user organization this means that existing systems have to be ported to the new hardware or software environment. Since some programming will be involved in that process anyway, modifications of the old programs to cope with new requirements might be added at the same time.

New technologies enable new solutions

An even more typical situation is that software vendors offer not only new technology but also new solutions based on that technology. That is, they provide new and better information systems than the old ones, using the new technology. For example, when SAP introduced the NetWeaver platform, a new IS architecture (ESOA – enterprise service oriented architecture) and new solutions to business problems that were not available before (xAPPS – SAP's version of packaged composite applications [Woods 2006]) were also introduced. User organizations that decide to migrate to the new technology can benefit from those new solutions.

Management involvement

Setting up an information system project costs money, and justification of that money is demanded throughout the project. *Management involvement* in a project is considered a critical success factor. The better the senior management's understanding of the project is, the better the expected results can be. A management standpoint like "I don't understand that technical stuff anyway" is not unusual but problematic for any IT project.

Creating awareness and justification can be difficult for the promotors of a project. If there is no clear understanding of the potentials and restrictions of information technology, then expectations are bound to be vague and sometimes exaggerated. Competing interest groups in the organization may try to influence the decisions of senior management regarding project acceptance, funding and planning in their respective directions.

Project idea is created by ...

Tasks to be solved and obstacles to be overcome in the process of obtaining project approval and funding depend on the degree to which senior management and departmental management were involved in creating the initiative for the project. A somewhat simplified differentiation of tasks is the following, as summarized in figure 2-1.

Senior management

1. The project idea was launched by *senior management*. In this case there is no need to create awareness of the problematic situation nor to convince the management of the necessity of the new system. However, before a project is agreed to, a cost-benefit analysis and estimates regarding expected project costs and duration will still be requested.

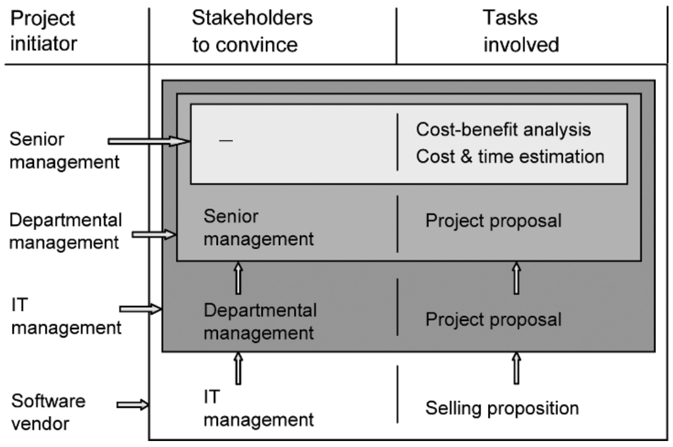
2. The project idea was born in a *department* of the business firm – or looking at a software company, by the marketing department looking for new business opportunities. In this case the senior management needs to be convinced of the project idea. A project proposal for the management will be produced in addition to a cost-benefit analysis and cost and time estimates.

Business department

3. The project idea originated in the *IT department*. This case has more obstacles to overcome. In the first place, the potential users in the company department(s) and their departmental managers have to be convinced. When the department's management is behind the project, then the senior management can be tackled. Project proposals may be needed for both the departmental management and the senior management. Costs, benefits and duration have to be assessed as in the above cases.

IT department

Figure 2-1 **Levels of project approval and tasks involved**



4. The idea for a new information system comes from a *software vendor*. This is basically the situation when a vendor seeks to sell a new product. Assuming that the user organization has an IT department, the first step might be to bring the CIO (chief information officer) and the IT managers on the vendor's side. IT personnel may have reservations about new systems and technologies

Software vendor

because they are used to the current situation and would require additional training. On the other hand, IT personnel tend to be interested in new technologies, so fewer problems can be expected from this side. Once the IT department is in line, the further steps are the same as in point (3).

2.2 Management Decisions

Typical decisions made by senior management are the decisions a) to set up a project, b) to redefine project goals and conditions after some time, and c) to continue or cancel a project, if it is not successful or for other reasons. In this section, the underlying decision problems are discussed.

Along with the decision to set up a project, management may be faced with another question: How and perhaps where should the project be done – develop the system inhouse, let a software firm develop, or buy from a software vendor? These questions will be addressed in section 2.3.

a) Setting up a project

Project proposal

Unless the initiative for a project was born by the senior management itself, the decision makers have to be convinced of the necessity of the project an/or the expected benefits. A common way to start the approval process is to write a project proposal. Such a proposal describes the objectives, the expected benefits, costs, risks, and the time frame of the project.

Management will evaluate the proposal against the business goals. Does the proposed information system match the firm's business strategy? Does it support the critical success factors? Which goals are better achieved if the project is successfully completed? Methods and techniques to answer those questions are available. Common approaches that have been used in practice for many years are business systems planning and information strategy planning.

Business systems planning (BSP)

Business systems planning (BSP) was initially developed by the IBM Corporation in the late 1960s and since then it has been continuously improved and extended [Zachman 1982]. The underlying idea is that information systems cannot be developed and operated in isolation. They need to be integrated into an enterprise-wide information systems architecture.

BSP provides a methodology to describe all data resources and all business processes of an organization and how they are interrelated. From such a description, individual information systems are derived and specified in terms of data and processes to be covered by the respective systems. While BSP is an approach to specifying the entire information systems landscape of an enterprise, the outcomes of a BSP study can later be used to evaluate a project proposal and determine the fitting of a new information system.

Information strategy planning (ISP)

Information strategy planning (ISP) is a part of information engineering, a methodology that James Martin made popular in the early 1990s. Information engineering is a very comprehensive approach to the planning, analysis, design and construction of information systems on an enterprise-wide basis applying semi-formal techniques and automated tools [Martin 1989].

Information strategy planning is the first of four information-engineering stages which finally lead to an interlocking set of running information systems in an enterprise. ISP covers the top management goals, the critical success factors, and how technology can be used to create new opportunities or competitive advantages. The outcome of ISP is a high-level specification of the enterprise's information needs and how they are related with the goals and the critical success factors. This specification can be used, like a BSP study, as a measure to evaluate a project proposal.

BSP and ISP start from scratch

Comprehensive methodological approaches such as BSP and information engineering started more or less from the "nothing is there" assumption, modeling the entire organization in terms of information technology concepts. However, with the dissemination of even more comprehensive standard software, the "nothing is there" assumption is not valid any more, thus decreasing the importance of BSP and information engineering substantially. Nevertheless, for an evaluation of how well a proposed information system would match the company's goals and critical success factors, it is extremely helpful if a high-level model of the respective relationships like an ISP model is available!

Another senior management decision in many cases is the "make or buy" decision. In such a case the project proposal will contain arguments both in favor of and against either one of the options.

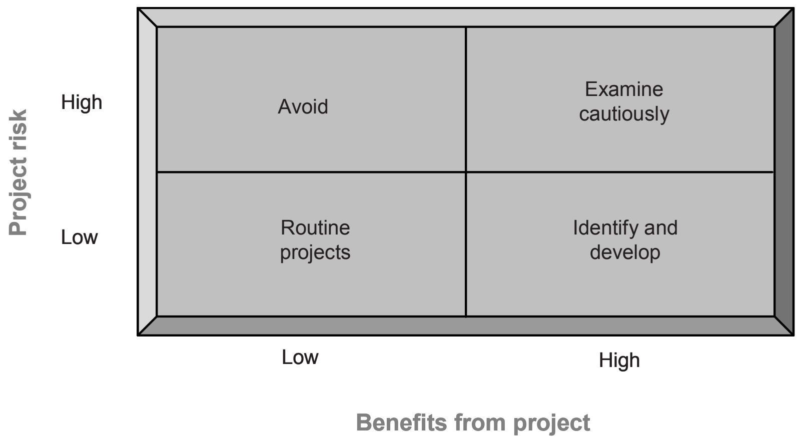
Portfolio analysis

Project portfolio

The option to produce a system inhouse may be subject to a *portfolio analysis*. A project portfolio is a tool to effectively identify, assess, select and manage a collection of projects. Portfolio analysis is particularly important for software organizations.

These organizations live off the returns from projects and may have many projects going on at the same time. Taking a new project into the portfolio is then based on a rating and an evaluation of *all* projects. One reason for this is that the company's resources have to be shared among the projects. For example, resources may need to be shifted from other projects to a project that has an urgent demand. Strategic factors playing an important role in the decision whether to start a new project or not are market share, market growth, project complexity, risk, expected cash flow etc.

Figure 2-2 **A project portfolio [Laudon 2007, p. 563]**



Criteria for portfolio analysis

If the organization is a *user organization,* there may also be several parallel IS projects competing for limited resources. Criteria that can be applied in a portfolio analysis are, for example, the risks of the projects, their benefits, how they match the firm's strategy, and how they fit the enterprise-wide information systems architecture [Cash 1992]. "High benefit/low risk/good strategy fit/good architecture fit" projects are the ideal projects; however, there may be other reasons and restrictions why non-ideal projects have to be included in the portfolio as well.

Project risk vs. project benefits

Figure 2-2 shows a simple project portfolio with the two dimensions "project risk" and "benefits from project". Projects in the lower right quadrant are the favorable ones. The organization should intensify and/or identify such projects. However, projects with high risk often promise high benefits as well, so they should also be taken into consideration and carefully examined.

Scoring models

Criteria and weights

When it is preferred either to buy an information system or to have it developed by a software company instead of developed inhouse, or more generally, when different alternatives are available, then so-called *scoring models* are a common way to arrive at a decision. A scoring model allows the decision makers to allocate importance to a criteria list by assigning weights to the criteria.

The problem underlying the example of figure 2-3 is the choice between two ERP systems. The company's decision makers did agree, for example, on higher weighting for order-processing related functions and on somewhat lower weighting for warehousing functions.

Criteria and weights

The % columns indicate how well the systems under discussion satisfy the company's requirements regarding the criteria list. For example, system A satisfies the requirements for order processing by 67 %, so A's score for that criterion is 268 (weight 4 x percent 67) while B gets a score of 292 (4 x 73). Assessing all criteria in the same way yields a total score of 3,128 for ERP system A and 3,300 for ERP system B, so B appears to be the better one for the organization.

Problem: agreeing on criteria

Compared to real-life scoring models, figure 2-3 contains only a very simple model. ERP systems, for example, have hundreds of functions, so the list of criteria is usually much longer. A difficult task is to find and agree on the really relevant criteria. Not only are the functional structures of different ERP systems quite different; what different people consider *relevant* criteria may also vary. Often far too many criteria are considered and given high importance. This is because it is difficult to image in advance what functions of a future system will really be needed unless the decision makers have thorough experience with systems similar to the ones under discussion.

Problem: qualitative factors

Another problem with scoring models is created by qualitative factors. Agreeing on the criteria to be applied and on appropriate weights for the criteria is a difficult problem.

Figure 2-3 **Example of a scoring model [Laudon 2007, p. 564]**

Criteria	Weight	ERP System A %	ERP System A Score	ERP System B %	ERP System B Score
1.0 Order processing					
1.1 Online order entry	4	67	268	73	292
1.2 Online pricing	4	81	324	87	348
1.3 Inventory check	4	72	288	81	324
1.4 Customer credit check	3	66	198	59	177
1.5 Invoicing	4	73	292	82	328
Total order processing			**1,370**		**1,469**
2.0 Inventory management					
2.1 Production forecasting	3	72	216	76	228
2.2 Production planning	4	79	316	81	324
2.3 Inventory control	4	68	272	80	320
2.4 Reports	3	71	213	68	207
Total inventory management			**1,017**		**1,079**
3.0 Warehousing					
3.1 Receiving	2	71	142	75	150
3.2 Picking/ packing	3	77	231	82	246
3.3 Shipping	4	92	368	89	356
Total warehousing			**741**		**752**
Grand total			**3,128**		**3,300**

Weights are the result of a decision process in which different stakeholders may pursue different interests. Similarly, determining the percentage to which a functional requirement is fullfilled may depend on highly subjective judgements.

Therefore, scoring models are used to support decision makers in their decision process rather than to substitute the decision as such.

b) Redefining a project

In the course of a project many unexpected things could happen: Cost or time may overrun, new technologies may emerge, a customer's requirements or priorities may change, the business strategy may shift, a system similar to the one under development may become available on the market, etc. Many projects eventually have to face a situation that they are challenged after some time.

Projects exceeding time and budget

A significant number of real-world projects are either delayed with respect to the project schedule, more expensive than expected, or both. The parties interested in the project and the project management are then under pressure to justify what has been achieved so far, to explain why it took longer or why it cost more than planned or both, and to argue for budget and/or time extensions. Decisions regarding the budget and the time frame are the senior management's responsibility. Based on project experience and reassessment of the risks, an adjusted cost and time plan will have to be approved.

c) Cancelling a project

Continue or cancel?

The pressure to justify an ongoing project may be so strong that the project stakeholders face the question: "Should the project be cancelled?" This is a difficult decision since time, money and human resources have been invested in the project. If the project is shut down then this investment, good will and trust in the developing organization are lost, and expected benefits will not be realized. The costs and lost opportunities of shutting down the project have to be assessed against the expected costs and benefits if the project is continued. It is again the senior management's responsibility to decide whether a project is continued or cancelled. Portfolio analysis can help to reach such a decision if the organization has several projects going on at the same time.

2.2.1 The Role of Specifications and Documents

Documents are the point of reference

Management decisions are based on documents and financial figures. Documents play an important role not only for senior management decisions but also for operational decisions and project management. Written documents are the points of reference for agreements with contractors, for the justification of project results, for assigning work to the project team, for project controlling and reporting, and for many more purposes. Important documents that serve as a basis for decisions by senior and operational management include the project proposal, the requirements specification, and various analysis and design models.

Project proposal

The purpose of writing a project proposal is to provide senior management with a comprehensive evaluation of the project to help them make their decision. Such a proposal should state:

- what needs to be done,
- why, when and how it should be done,
- who is responsible and who is going to do the work,
- how much will it cost,
- what are the benefits,
- what are alternatives,
- what are the risks?

Contents of a project proposal

Although the structure and contents of a project proposal depend on the specific problem situation and on the organization's requirements, a typical proposal may contain an executive summary and sections like the following [EFC 2006]:

Needs statement

- *Needs statement:* It should be a concise, yet convincing overview of the needs the organization wants to address with the project. The reader should get a complete picture of the scope of the problem. How important is the project, and what are the consequences if the project is not carried out?

- *Goals and objectives:* This section should make clear to the senior management which business goals and critical success factors will be supported by the new information system. It will also define the specific goals of the project (e.g. reduce inventory costs by x %) and the objectives, i.e. specific, tangible and measurable outcomes that should be achieved within a specified period of time.

Goals and objectives

- *Approach and timetable:* How and when are the project's objectives going to be achieved and by whom are the primary questions addressed in this part of the document. It comprises a proposal regarding the "make or buy" decision, or an evaluation of these two alternatives. The "how" question is further refined by a rough project plan composed of the major sections of the project. Each section is terminated by a milestone ("milestone plan"). The human resources involved in each section are specified.

Approach and timetable

- *Cost-benefit analysis:* The costs caused by the project are specified, and expected benefits are elaborated. Benefits can either be short-term or long-term. Some benefits can be measured in financial figures (e.g. 10 % savings in transportation costs, 15 % additional revenue from faster delivery to retail stores) while others are qualitative benefits requiring causal analysis and argumentation (e.g. better service level).

Cost-benefit analysis

- *Project budget:* A budget summary states the duration of the project and the total project cost, as well as any already available income. There are different ways to structure a budget depending on the type of the project and on the organizations's requirements. However, almost every budget includes items like: project personnel, software costs (licenses etc.), additional hardware requirements and other equipment, traveling, meetings, training the future users, and overhead costs such as project administration.

Project budget

- *Project risks:* A description of the major risks of the proposed project is an essential part of the document. Project risks can originate from the task to be solved (e.g. too complex, too many departments involved), from project management (e.g. vague time estimates, inappropriate development tools), from the project team (e.g. competencies, knowledge level, experience etc. of team members, team size), from the IT infrastructure, from the implementation process in the organization (e.g. acceptance by users), and also from senior management (e.g. lack of support inside the organization).

Project risks

Project controlling and evaluation

- *Project controlling and evaluation:* This section should state how the progress – success or failure – in reaching the stated objectives is measured. Who will conduct the evaluation, when will it be performed, and how will the reporting be done?

Future costs

- *Future costs*: A statement of the financial and human resources needed in the operation of the information system once the project is completed and the system is implemented should be included in the document, e.g. resources for maintenance of the system, for user support and training, for software upgrades etc.

The project proposal has to be convincing because the main purpose is to obtain approval for the project by the decision makers. Equally important is the budget and/or the allocation of resources to be granted for the project. Often the approval and the budget are limited to some initial project stages. The decision to continue the project will be made at a later time, based on results achieved or insights gained by that time. Milestones serve the purpose of evaluating project progress and deciding whether to continue, cancel or reshape the project and whether to re-allocate resources.

Requirements specification

Requirements engineering

Once the project is approved, the requirements that the information system is expected to satisfy have to be elaborated in more detail. This is the subject of the *requirements engineering* stage in the course of the project. Requirements determine the outcome of the project. If the requirements are not right then the resulting information system will not do what the stakeholders expected. Requirements engineering is a particularly critical stage in most projects and known to be difficult. It has evolved into a discipline of its own that will be discussed in chapter 5.

Requirements specification document

The outcome of requirements engineering is again a document (or a collection of documents). It is called either a requirements specification, requirements document, or software requirements specification (SRS).

Point of reference

This document may serve as a reference for different purposes and for different types of users. In the "buy" case, requests for quotations may be issued, and quotations received may be evaluated based on that document. Likewise an agreement with an external contractor to build the system will refer to the requirements specification. If the system is built inhouse, then the specification is the document that is given to the system-development group as the starting point for their design considerations [Sommerville 2007, p. 137].

In the implementation and testing stages, the requirements specification is used by the test group to develop validation tests for the information system, and to test the system against initial requirements. Even in the later operations and maintenance stages, the requirements specification can be used by support and maintenance personnel to better understand the system and the relationships between its parts.

Figure 2-4 **A requirements specification**[§]

Chapter	Description
Preface	Define the expected readership of the document and describe its version history.
Introduction	Describe the need for the system. Describe its functions and explain how it will work with other systems. Describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	Define the technical terms used in the document. Do not make assumptions about the experience or expertise of the reader.
User requirements definition	Describe the services provided for the user and the non-functional system requirements – in natural language, diagrams or other notations that are understandable by customers. Define the user interface (forms, menu structure, navigation). Product and process standards which must be followed should be specified.
System architecture	Present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements	Describe the functional and non-functional specification require-ments in more detail. If necessary, further detail may also be added to the non-functional requirements, e.g. interfaces to other systems.
System models	Set out one or more system models showing the relationships be-tween the system components and the system and its environment (e.g. object models, data-flow models, semantic data models).
System evolution	Describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	Provide detailed, specific information which is related to the applica-tion which is being developed. Examples are hardware and data-base descriptions (e.g. minimal and optimal configurations for the system).
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

§ Adapted from: Sommerville 2007, p. 139.

IEEE standard 830-1998

Guidelines on how to create a good requirements specification are provided by professional societies, software organization, consultancies etc. An often cited document is the "IEEE recommended practice for software requirements specifications" (IEEE standard 830-1998) by the Institute of Electrical and Electronics Engineers, Inc. (IEEE). Unfortunately this document is provided only for IEEE members. Guidelines are also found in most books on software or requirements engineering. An example adapted from Sommerville's book is given in figure 2-4.

More documents and models are used in later stages of a project. In the design stage, for example, models play an important role as specifications and reference documents for software developers. (Modeling techniques are discussed in detail in chapter 5.) In the testing stage, test plans are created and documented in formal test specifications (cf. section 6.3.2).

2.2.2 Milestones and Deliverables

For controlling a project's progress and for judging whether the project is on the right track regarding its objectives and schedule, appropriate information is indispensable. Managers facing requests for more resources or having to decide on whether to continue or cancel a project need tangible results as a basis for their decision.

Project milestones

Milestones are distinct points in a project where a project activity or a work package ends. Major milestones include the end of each logical stage in the project according to the underlying process model. When the milestone "design stage completed" is reached, for example, the activities of the next stage (implementation) can be launched. Typically, once a milestone is achieved, it is set out in a document – the milestone report – and associated with some sort of decision about the future of the project. An example of milestones that a software company might define for the later stages of a standard-software development project is given in figure 2-5.

Milestones should be operational

Achievements expected at a major milestone should be specified in the project proposal already. In this way senior management can see what the potential breakpoints in the project are. It is important to define milestones that are operational, i.e. successful completion of the respective activity must be measurable or at least demonstrable. A milestone like "90 % of the code is ready" is useless since plain code volume says nothing about the time needed to accomplish a working system.

Figure 2-5 **Milestones for second-half project stages [Rothman 1997]**

Milestone	*Criteria*
Feature freeze	All required features of the system are known and the detailed design has uncovered no more. No more features are inserted into the product.
Code freeze	Implementation of the design has stopped. Some testing of the features has occurred.
System test freeze	Integration testing is complete. Code freeze for system test to start.
Beta ship	The date the Beta software ships to Beta customers.
Product ship	The date the product ships to the general customer base.

Mini-milestones

For work-assignment and project-management purposes, milestones can be broken down into *mini-milestones*. While the major milestones reflect high-level activities of several weeks or months duration, mini-milestones take less than one or two days effort and are measured in hours. The advantages are improved status reporting, fine-grain control of knowing if a mini-milestone is missed, improved motivation because every day or so a mini-milestone is achieved, and reduced schedule risk [Perks 2003].

Deliverables

Deliverables are intermediate or final project results that are handed over to the person or organization that gave the order for the project, e.g. the customer (external) or the organization's management (internal). Typical deliverables are documents produced at the end of a project stage or a work package such as a design specification, a system prototype or a test report.

Milestones ≠ deliverables

Milestones and deliverables are related but not identical. Milestones may be associated with deliverables. Deliverables are results intended for the people the project manager (or another person in a position of responsibility) is reporting to. Milestones are points where major steps are completed. Results may, but not necessarily, be of interest to customers or top managers. Some results are important just for the project management, serving as internal milestones within the project.

2.2.3 Build, Buy or Rent?

Many ways to obtain an IS

As we pointed out at the beginning of this book, there are many ways to obtain an information system, i.e. building the system inhouse, buying and customizing standard software, having a domestic or foreign (e.g. Indian) software firm develop the system, obtaining desired system services on demand, etc. Deciding which of the different ways to choose is the senior management's responsibility.

All options have advantages and disadvantages, sometimes beyond the information system needs at hand. For example, it may be cheaper to give the order to develop the system to an external software firm, yet this also means that expertise gained in the development process will not be inside the company but outside. If this path is repeatedly chosen, then less and less development knowledge and knowhow will be retained internally. As a consequence, the company will depend on external software firms for future developments as well.

SMEs have fewer options

Being able to choose among alternatives requires, of course, that the mentioned alternatives are available to the organization. A small or medium-size enterprise (SME) that has no software development personnel obviously does not have an option to develop the system inhouse. Therefore it is large companies that can select from the full range of possibilities. However, small and medium-size companies that develop their own software also exist, in particular if they are highly specialized and need specialized software.

Below we will discuss the options of developing, buying and renting an information system. In the case that the system is not bought off the shelf but has to be developed, we can distinguish further who is in control of the development process.

1. Developing inhouse

Conventional information systems development

Inhouse personnel developing the complete system has been the conventional way through which many information systems have come into existence. Traditional approaches to ISD have mostly assumed that the system under consideration is built inhouse. Models and methods available for this case are discussed in chapters 4 to 6.

2. Developing with external partners

Working with a software firm

Instead of developing inhouse, the company may decide to have either the entire or parts of the system built by an external contractor. Usually the contractor is a professional software firm whose core business is software. Reasons to place an external order are manifold: The user company may not have the manpower to develop the system; technical knowhow may be lacking; the software firm is known to have experience in a specific field; external development is less expensive than internal; and more.

Large projects may be split up into parts. In this case, some parts can be built inhouse while others are ordered from one or more external partners. In the past, the conventional way of placing an order was to commission a domestic software firm. Today, in a globalized world, competitors from different countries and continents are offering their services. When the contractor resides in a different country, in particular in a low-wage country, then the development order falls into the category of *offshoring* (see below).

Requirements specification as an interface

Overall control of the development process and the external orders remains with the user organization. Since a division of labor between the company placing the order and the company completing the contract is involved, a clearcut interface between the two is needed. A typical interface is the *requirements specification* as described in section 2.2.1. This means that the initial work is done by the user organization – in particular defining the information system's scope, elaborating requirements, and describing the requirements in a document. The requirements specification is used to evaluate intermediate and final results provided by the contractor, and to accept or reject the delivered system.

Design specification as an interface

External partners may also be commissioned in later stages of a large project. Sometimes the system design is made by the user organization, in addition to the requirements specification, but the implementation according to the *design specification* is given to a software firm with expertise in the software technology required for the coding. Some organizations even give the testing of a software system to external companies that are known system testing specialists.

3. Ordering an individual turnkey solution

External partner has complete responsibility

While in the previous case the user organization remains in full control of the total process, an organization may prefer to be relieved of that burden. This can be the case, for example, in small and medium-size companies where the knowhow and manpower to perform an IS project are not available. Under the assumption that an individual solution is necessary and standard software is not available for the problem at hand, such a company would rather have an external specialist do the entire development process.

In this case, the initial project steps also have to be taken by the contractor. In particular, creating an initial project document will be a task of the contractor. This document should contain topics similar to a project proposal, as far as that they are applicable, such as objectives, timetable, milestones, costs, benefits and future costs. The requirements engineering process will also be conducted by the contractor. This means that people from the software firm will go into the company, study processes and documents, interview employees to elicit user requirements etc., in order to create a requirements specification.

Subcontracting

It should be noted that an external contractor is in a similar situation as the initial company in which the information system need arose. The contractor may develop the ordered (sub-) system inhouse or commission subcontractors for parts of that system. In large projects subcontracting is a common practice.

4. Buying, customizing and extending standard software

A common way for a user organization to obtain an information system is to buy or license standard software which is available for many business problems. This approach is discussed in detail in chapter 7. It is quite popular because standard software has many advantages. A major one is that standard software is usually cheaper than the development of an entirely new information system from scratch.

Adaptations and extensions

As pointed out earlier, standard packages rarely meet all individual requirements of a particular organization. In general, they have to be adapted to the organization's needs (customization). Missing features, i.e. functionality that is not contained in the standard package, must be provided. Additional information systems or add-ons to the standard software have to be developed for that purpose – by the package vendor, by the company itself, or by external contractors.

5. Employing an application service provider (ASP)

Application service providing is a business model in which a company, the application service provider (ASP), makes computer-based services available to other companies (customers). Application service providing can be seen as a value-adding continuation of the outsourcing of hardware resources and operations to dedicated partners. Formerly hardware outsourcing meant that instead of maintaining hardware capacity within an individual organization, a remote computing center operated by a specialized provider is employed. The customer would then run their programs on the provider's hardware.

Importing computing services

In the ASP model, not only are customers provided with hardware or computing power in the narrow sense, but they are also provided with *services*. Such services are wide ranging, from specialized functions like invoicing, tax calculations, online payment and credit card processing via all-in-one coverage of the IS needs of a particular industry or profession (e.g. lawyers) to comprehensive information system support for small and medium-size businesses. Complete application packages for financial management, enterprise resource planning, customer relationship management etc. are available from ASPs. Collaborating with an ASP can be an attractive option because the company does not need to install, operate and maintain complex information systems itself in order to receive a guaranteed service and support level.

In the marketplace, there are thousands of ASPs for a large number of application problems. ASPstreet.com, for example, a web portal for application service providing, lists about 4,100 ASPs [ASPstreet 2006]. Big players like IBM, SAP and Microsoft are engaged in application service providing.

ASP examples

IBM, for example, has offered business services for a long time in areas such as financial, human-resources, supply-chain and customer relationship management. As an application service provider, IBM makes software from Oracle, SAP, Peoplesoft etc. available to other companies. SAP targets the small and medium-size enterprise market with its All-in-One software and SAP business partners acting as ASPs. Microsoft, as another example, offers its Commerce Manager software for creating online stores to small businesses. A well-known ASP in the field of customer relationship management and related areas is Salesforce.com [Salesforce 2006].

Customers pay for software usage

Customers of an ASP pay for the *use* of the software, not for a license. Since the provider serves many customers with the same software, individual fees for employing the service can be quite low. A number of different payment schemes are in use, for example subscription based (e.g. monthly fee) or per transaction.

ASP software does not fit all

The ASP model works well if software exactly fitting the organization's needs is available. Since this is rarely the case for complex business problems, customization needs can be a major obstacle. The ASP is more likely to customize its software for large customers than for a variety of potential small customers. Another problem is integration with the customers's other information systems. If the software provided by the ASP requires data, for example, that are distributed in the company's database, then that company has to fill the gap by bridge programs retrieving and preparing the data in the required format.

2.3 Global Options: Outsourcing and Offshoring

Software development is not the core business of most companies, it requires expertise and manpower, and above all it is rather costly. Therefore it is not surprising that many organizations prefer to use other companies which are experienced in software development.

Driving factors are price and quality

Since development costs are significantly lower in Asian, Latin American and Eastern European countries than in the United States and in Western Europe, many software orders have gone to vendors in these regions. While price has been the driving factor for many years, other reasons have also emerged in the recent past. For example, the know-how and maturity levels of professional software companies in India are on average higher than in the US and Western Europe.

Domestic and global options

Nowadays, organizations that do develop software – user organizations and software companies as well – have several choices. They can choose between inhouse and external development. If external development is the preferred option, the next question is whether the external partner should be domestic or foreign. In the case of a multinational company, another choice is between developing at home vs. developing in a location abroad where that company has a branch and where software development is more cost-effective. It is the management's task to decide which of the various alternatives to choose.

A number of terms have been coined for the modes of information systems development abroad, including offshoring, nearshoring, and offshore outsourcing. We will start with a brief look into some terms related to the location of development.

Outsourcing

Started in the 1990s

Outsourcing is a business practice that became popular when a general reorientation of business strategies took place in the 1990s. Many companies put a stronger focus on their core competencies and transferred non-core activities from internal business units to external organizations such as subcontractors specializing in those activities. One of the first spectacular outsourcing deals in information technology was the Eastman Kodak deal, which resulted in that company's entire data center operations being outsourced to IBM, Digital Equipment and Businessland in 1990.

This example shows that outsourcing is not a specific practice for software development but possible for any non-core business function. In the IT field, outsourcing deals cover a wide range, from software development via processing transactions in dedicated application areas (e.g. banking, insurance, flight reservations) to outsourcing the company's complete IT infrastructure.

Offshoring

Offshore, onshore and onsite

The notion of offshoring is mostly used in the context of information technology although its general meaning is just to do something "off" one's own "shore". With respect to software development, three related terms are onsite, onshore and offshore development. While *onsite* means development at the organization's location, *onshore* stands for development at a different place in the same country (e.g. by a domestic contractor), and *offshore* stands for development in a different country.

Types of work sent offshore

Offshoring is a concept that comprises several operational models for all kinds of IT-related activities. A task force of the ACM (Association for Computing Machinery) distinguished between six different types of work sent offshore [Aspray 2006, p. 19]:

1. Programming, software testing, and software maintenance
2. IT research and development
3. High-end work such as software architecture, product design, project management, IT consulting and business strategy
4. Physical product manufacturing (semiconductors, computer components, computers)
5. Business process outsourcing (e.g. insurance claim processing, accounting, bookkeeping) and IT enabled services (e.g. financial analysis, reading x-rays)
6. Call centers and telemarketing

Organization of offshoring

With regard to the topic of this book, information systems development, the first and the third categories are the ones to consider. Looking at the companies offshoring software-related work today, the following models can be identified:

a) *Captive centers:* Many multinational companies employ local branches in low-wage countries for software development or set up specific development centers. SAP, for example, develops significant parts of its business software in its Bangalore branch in India. American Express, Citibank, Dell, Continental and many other big companies run captive IT centers abroad.

b) *Joint ventures:* The company interested in offshoring and an outsourcing provider enter into a joint venture regarding IT services. The two parties set up a new firm that will carry out development projects.

c) *Offshore outsourcing:* A third party – usually a software firm in a low-wage country providing outsourcing services to other companies – is contracted by a customer to develop one or more information systems (or for other IT services). Novices in offshoring can seek help from domestic firms offering brokerage services.

d) *Global IT partners:* Large IT companies in offshore countries have entered the world market, offering their services onshore. This means that a customer outsources work to a domestic branch of the foreign IT company who in turn sends parts of the work to their development centers at home. Indian companies such as TCS (Tata Consultancy Services), Infosys, Wipro and Satyam have become important players on the world market in this way.

Captive centers and joint ventures are the most common forms

Big offshoring deals covered in the media usually involve a captive center or a joint venture. On the other hand, outsourcing to a different offshore company, without face-to-face contact and physically separated by thousands of kilometers, is not a widely used practice. However, outsourcing information systems development to an external partner is a viable mode in the following:

– Global IT firms offer their services onshore (case 'd' above)
– When the customer has a branch in the offshore country
– When the offshore company has a country office based onshore
– When a reliable broker is available

Nearshoring

Nearshoring is a variant of offshoring in which the "shore" is nearer than India or China. Nearshoring means relocation of activities to lower-cost foreign locations, but in close geographical proximity. For the US, typical destinations of nearshoring are Mexico and Canada; for Western European countries, Eastern Europe is a favorite location.

Cultural differences are smaller

Reasons why nearshoring is preferred over offshoring include cultural closeness and sometimes fewer language problems. Germans, for example, may feel more comfortable working with people from Slovakia or Hungary than with people from China because cultural differences are smaller, and they might even be able to communicate in German.

Global IT service providers

It should be noted that the borders between offshore, nearshore, onshore and even onsite are blurring. Offshoring and nearshoring providers are moving into the domestic markets with their own branches. The goal of such a move is to enhance the provider's competitive position. An offshoring provider with a branch in Germany, the UK or the US can present itself as a domestic software company in that country – with a significant cost advantage over their competition because they can give labor-intense work to the mother company in India or China.

Nearshore "front-ends" to offshoring

Likewise it has been observed that offshoring providers have opened branches "near shore". Knowing that many user organizations prefer nearshoring over offshoring ("farshoring"), the provider opens a branch or establishes a joint venture either near the US, UK or Germany. In this way the offshoring provider can act as a nearshoring provider and attract customers that are willing to outsource nearshore but not farshore.

2.3.1 Offshoring Strategy

In this section, the aims, mechanisms, benefits, risks and costs of offshoring are discussed. Most organizations that have partially or completely offshored their information systems development consider offshoring a long-term strategy and not a one-off occurrence conducted for the sake of a single information system. Establishing this strategy is the first step before all other activities can begin. This includes a number of subtasks. While the first subtask occurs in all four types of offshoring, the second one is specific to offshore outsourcing. The third subtask described below refers to captive centers and joint ventures.

I. Selecting the country

This is an important decision because capabilities, risks and benefits differ between countries. Different countries may be suitable for different aims and purposes. Factors to consider are political stability, infrastructure, size and quality of the IT labor pool, language and cultural issues, data security, protection of intellectual property rights, software piracy and government support.

India is the market leader

India is by far the market leader in offshore IT work, followed by China, Canada, Mexico, Ireland, Malaysia, South Africa, Israel, Russia and the Philippines [Aspray 2006, pp. 52-54]. More than half of the offshoring deals worldwide reported go to India. Eastern European countries are increasingly considered for nearshoring by companies in Western Europe.

Qualified Indian IT graduates

India's predominance is based on several facts, including an education system that has placed great emphasis on computer science and mathematics, generating a large number of graduates in the field of information technology every year. Although English is not the mother-tongue of most Indians, it is widely used in higher education. Most computer science and business graduates speak fluent English, so the language barrier for communication with customers in English language countries is low.

CMMI levels 3 and higher

Since IT services and outsourcing have been big business in India for many years, the number of qualified providers exceeds by far the numbers in other countries. The level of knowledge and experience is very high. More Indian software companies are certified as levels 3 and higher in the capabilities maturity model (CMMI)[§] than in any other country, including the US and Europe. One reason for this is that the Indian government started to strongly support the IT industry in the 1990s, including deregulation and liberalization, and providing numerous incentives such as tax exemption for IT enabled services.

Due to the high Indian maturity level, offshore costs have risen, and other countries are becoming more competitive. It has been reported

§ CMMI, originally the SEI-CMM (capability maturity model) was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. It is a framework for evaluating and measuring the maturity of the software development process of an organization on a scale of 1 to 5. Those levels describe stages on an evolutionary path from an ad hoc, immature process to a mature, disciplined process. Key practices are defined in the CMM, intended to improve the ability of organizations to meet goals for cost, schedule, functionality and product quality. The SEI substituted the CMM by CMMI (capability maturity model integration) in 2000 [SEI 2007].

that Indian IT companies are now subcontracting software firms in Bangladesh, Sri Lanka and Nepal due to lower costs.

II. Selecting the vendor

In the case of offshore outsourcing, an appropriate partner in the offshore country has to be selected. This involves the following steps [EBS 2006]:

a) *Examining the vendor landscape:* In countries providing offshoring services, there are usually a large number of potential partners. If no previous experience with any of those partners is available, screening criteria have to be employed, for example experience, domain expertise, cost, quality and financial stability.

b) *Determining the cooperation model:* The best-fit model for the cooperation has to be selected. Mature outsourcing providers offer not only information system development but also other services, for example support, helpdesk and even operating the system afterwards.

Criteria

c) *Narrowing down the vendors:* Criteria may include years in business, scale of operation, range of services, geographic span, delivery model options, industry focus and cultural fit. Experienced outsourcing specialists recommend visiting each vendor to personally assess each key performance criterion that is considered important to the company.

Negotiations

d) *Negotiating contract and relationship:* The company finally has to negotiate and attempt to build a relationship with the selected vendor. Since this is a strategic relationship that might be difficult to exit later, taking the time to negotiate suitable terms and conditions is very important.

III. Establishing an offshore center

Captive centers, joint ventures

Different steps are necessary when the offshoring provider is a captive center or a joint venture. In this case, the strategic division of tasks between the organization at home and the offshore center has to be decided: Which tasks will be sent offshore, which ones will remain onshore? These questions are discussed below (cf. critical issues).

As in the offshore outsourcing case, contractual relationships between the onshore and offshore organizations have to be established – for example: How will the offshore center accept work (via quotation, statement of work, order, contract)? How will the internal cost be allocated and charged (e.g. who will invoice whom, how will internal service charges be determined)?

Critical issues

Transition to offshore

Apart from vendor selection, there are critical issues to be considered in the strategy stage. Organizations with existing development teams have to prepare the transition to offshoring very carefully. Onsite developers may fear the loss of their jobs and refuse to cooperate. Offshore personnel will then fail to receive necessary information which is needed in order to understand the customer's processes and requirements, and mixed onsite-offshore teams will not work well. Expected benefits of offshoring will not be realized in such a case.

Intellectual property concerns

Intellectual property rights may be a serious concern for some companies. "What if source code developed for me is also sold to my competitors? What if an employee of the service provider steals code and sells it to my competitors?" are two of the questions asked [Tatvasoft 2006a]. Appropriate legal provisions and technical measures to prevent violation of intellectual property have to be established.

2.3.2 Offshoring Projects

Information systems development projects that are most suitable for offshoring are projects that automate well-documented business functions or processes where little day-to-day interaction is required. An ideal development task would be one that is completely specified in terms of the process steps, inputs and outputs. In such a case, a specification could be "thrown over the fence" (i.e. handed over to the offshoring partner) and an information system will be delivered as a result of the project.

Onsite and offshore personnel must cooperate

Unfortunately most projects are not of that nature and require a lot of interaction. Therefore project teams have to be set up in a way that onsite and offshore personnel communicate intensely. Most projects have offshore personnel working onsite for that reason.

Process models also have to be adapted to offshoring requirements. Process models for offshoring projects will be discussed in section 4.5.

Critical issues

Manpower turnover

Offshore software companies often hire people (e.g. freelancers) for new projects, or subcontract other software development firms. The customer may also have their own development staff involved in the project. All these different people have to work together smoothly. Since staff turnover in the booming offshoring industry in India is high, new people may have to be integrated into the team rather frequently. These issues create significant challenges for project management.

Different time zones

Different time zones may generate problems and frustration in communication. When the customer in New York City, for example, sends an urgent change request at lunchtime to the development team in Pune, India, it is almost midnight there, and the request is not likely to be processed until the next day. Some offshore companies try to cope with this situation by working according to the customer's hours or at least ensuring communication 24 hours a day.

Offshore developers not available later

Maintaining, changing and enhancing the information system once the project is completed (i.e. the system is installed and running at the customer's site) must also be planned on time. Offshore developers may no longer be available, having moved on to other projects or employers. Having onshore developers working in the project team can therefore help to reduce after-project problems, because these developers will have the same knowledge about the system as the offshore developers.

2.3.3 Benefits, Pitfalls and Risks of Offshoring

Offshoring has become a common option for software development since significant benefits are associated with it. On the other hand, failures have also been reported, and many organizations are hesitant to start offshoring because of the risks. These advantages, problems and risks are discussed below.

Benefits

Cost savings

1. *Cost savings:* Most companies that started offshoring projects were initially attracted by obvious cost savings. Salaries of skilled software developers in India some years ago were about 20 % of

the salaries in the US and Western Europe. Having people with 80 % lower pay work on a project does not mean, however, that the savings will be 80 % compared to an onsite or onshore project. Additional and hidden costs accrue, so that the total savings are a lot lower. Offshoring projects report total cost reductions of 15 to 40 % compared to onsite projects, which is nevertheless a significant saving [Davison 2003].

Higher quality

2. *Quality:* In established offshoring locations the quality level has risen quite high. The quality of the information systems obtained is another and increasingly important factor why offshore outsourcing is practiced. India, for example, is loosing its cost advantage because of rapidly growing IT salaries. However, professional Indian IT companies are on average capable of producing higher quality software than domestic software firms – at least regarding certification according to the ISO 9000 standards and the CMMI levels. Customers in Europe and America appreciate the high quality they get.

Availability of IT skills

3. *Availability of IT skills:* Offshore IT companies and captive centers can provide well-trained software developers experienced in cutting-edge technologies that may not be available onsite. Building up knowledge and expertise inhouse may be much more expensive than employing knowledge and expertise of personnel provided by an offshore company. Time-to-market is shorter when experienced staff are available.

Availability of resources

4. *Resources:* Organizations lacking manpower for software development can overcome their shortages through outsourcing projects. Outsourcing permits an organization to free resources and appoint available personnel to high-priority or greater-value adding activities.

Customer orientation

5. *Customer orientation:* Nowadays, offshoring providers have to withstand stronger competition. The offshoring portal OffshoreXperts.com lists more than 50,000 offshoring providers in the IT field [Offshore 2008]. Therefore it is not surprising that most offshoring providers have a sound customer orientation.

Working morale

6. *Working morale:* Being customer-oriented, outsourcing providers may be more willing to satisfy the customer than an internal development group might be. Higher flexibility, desire to meet deadlines, and quick responses to customer requests can be seen. "For last minutes changes, we don't turn the light off at five" is a slogan on the website of an outsourcing provider [Tatvasoft

2006c]. Captive centers in an offshore country also enjoy the working morale common for that country.

Pitfalls and risks of offshore outsourcing

While some critical offshoring issues were mentioned before, we now look at the problems and risks of offshoring in more detail. Particular attention will be given to offshore development projects in the next subsection. Immediate and hidden costs of offshoring will be discussed in the section 2.3.4.

Unrealistic a priori expectations

Offshoring strategies and/or projects can fail, yet there is no indication that offshore projects show a greater failure rate than onshore projects. What exactly is considered a failure depends on *a priori* expectations. Assumptions regarding the cost savings, for example, may not be fullfilled because they were unrealistic. The table of risks shown in figure 2-6 is headed by erroneous cost expectations. The list was published by Meta Group (now part of Gartner Group) as a top 10 list of the risks related to offshore outsourcing. Figure 2-6 gives a summary of those risks.

Underestimating the complexity of setting up and managing an offshore project is another pitfall. Not only the geographical distribution but also cultural and language differences can make an offshoring project difficult to manage. Even if the customer's language is the same as the outsourcing company's, misunderstandings and problems occur because of social, religious and behavioral differences. The legal environment, civil rights, bureaucracy and an unstable political situation in the offshoring country are further sources of risk.

Impact on inhouse personnel

Interfacing the offshoring provider with the remaining organization, in particular with inhouse software developers, is a serious management challenge. Those who "survive" the partial outsourcing of software development might still not be supportive of the deal and resist cooperation.

When more stages of the software life cycle are outsourced, more people in the organization are affected. When coding and testing are outsourced, ordinary programmers become dispensible. When design is also outsourced to the offshoring provider, software architects are affected. When requirements analysis and definition are outsourced, systems analysts are not needed to the same extent as before.

Outsourcing life-cycle activities does not mean that all onsite personnel who previously did the respective jobs are laid off. People closer to coding and testing are affected more than personnel further up in the life cycle.

Figure 2-6 **Risks of offshore outsourcing [Davison 2003]**

Top 10 Risks of Offshore Outsourcing	
1. Cost-reduction expectations	Executives assume that labor arbitrage will yield savings comparable to salary differences without regard for the hidden costs. In reality, most organizations save 15 % - 25 % during the first year; by the third year, cost savings often reach 35 % - 40 % as companies go up the learning curve.
2. Data security/ protection	The vendor might not have sufficiently robust security practices or might not be able to meet the company's internal security requirements. Security breaks, intellectual property and privacy violations may occur.
3. Process discipline (CMM)	The company is lacking internal process model maturity. Meta Group observes that appr. 70 % of IT organizations are at CMM level 1 while many offshore vendors' characteristic is level 5. This will undermine potential cost savings.
4. Loss of business knowledge	Most organizations have business knowledge that resides within the developers of applications. In some cases, this expertise may be a proprietary or competitive advantage. Companies must carefully assess business knowledge and determine if moving it offshore will compromise company practices.
5. Vendor failure to deliver	A common oversight is a contingency plan – what happens if the vendor, all best intentions and contracts aside, simply fails to deliver. The organization should assess the implications of vendor failure (i.e., does failure have significant business performance implications?).
6. Scope creep	Most projects change by 10 - 15 % during the development cycle. Organizations are surprised that the vendor expects to be paid for incremental scope changes.
7. Government oversight/ regulation	Organizations facing government oversight (e.g. healthcare) must ensure that the offshore vendor is sensitive to industry-specific requirements; able to comply with government regulations; and accountable during audits.
8. Culture	A representative example: although English is one official language in India, pronunciation and accents can vary tremendously. Cultural differences include religions, modes of dress, social activities, and even the way a question is answered. Executives should not assume that cultural alignment will be insignificant or trivial.
9. Turnover of key personnel	Rapid growth of outsourcing vendors has created a dynamic labor market with high demand for key personnel. Turnover levels are in the 15 % - 20 % range. The impact of high turnover has an indirect cost, increasing the time spent on knowledge transfer and training new individuals.
10. Knowledge transfer	The time and effort to transfer knowledge to the vendor is a cost rarely accounted for. We observe that most organizations experience a 20 % decline in productivity during the first year of an agreement, largely due to time spent transferring both technical and business knowledge to the vendor.

Project managers, for example, are still needed, yet the focus of their activities is shifted from cordinating technical staff and detailed development activities to coordinating onsite and offshore activities and people.

Shift of skill profiles required

The management's challenge is to communicate the benefits of offshoring for the company's competitiveness to their staff and to manage the transformation process from inhouse to offshore development. In offshoring projects, there is still plenty of work left to be done in the customer's organization. Yet this work is different, because it is focused on the business and departmental level.

An example is preparing projects to make them ready for offshoring and identifying new opportunities for IS solutions. Personnel further down the development cycle may be qualified to take on work higher in the life cycle, closer to the business problems, or in the coordination of onsite and offshore activities. Such activities are discussed in more detail in section 4.5.1.

Risk management is important

Many companies fail to manage risks. A proper risk assessment and mitigation plan should be prepared in advance [Morrison 2005]. Infosys, a leading Indian outsourcing provider, includes a detailed plan for risk identification, monitoring and mitigation as part of project planning. This plan covers risk identification, prioritization and mitigation options. The status of the risks is continuously tracked and reviewed using a monthly milestone mechanism [Infosys 2008].

The bottom line is: As organizations consider the vast benefits and allure of offshoring, they must also balance the risks and uncertainties with the potential for labor arbitrage [Davison 2003].

Risks of offshore development projects

Any offshore software development bears a number of risks, no matter whether it has been outsourced to a different organization, a captive center or a joint venture under the control of the customer. Among the risk factors are the following [Sakthivel 2007]:

Collaboration management and tools

- *Coordination of collaborative work:* Teams composed of onshore and offshore staff need to collaborate effectively over large distances. Face-to-face interaction and meetings have to be substituted by online collaboration tools (groupware, project repository, videoconferencing etc.). Problems can arise due to the shortcomings of the tools, incompatibilities between different tools and lack of acceptance by the project members. The less powerful and the less integrated the tools are, the more face-to-face interaction will be required, implying time-consuming journeys, costs and delays.

Quality of documents

- *Quality of requirements and design specifications:* Requirements engineering is based on interaction with the stakeholders, and likewise, deriving a good design needs interaction with and feedback from the requirements engineers. These things are difficult to do over a distance, with the help of electronic means only. Creating and communicating appropriate requirements and design specifications in this way bears a significant risk of misunderstanding and misinterpretation. When onsite personnel create the specifications, these documents should be clear and unambiguous. However, from requirements engineering we know how difficult precise requirements specifications are.

Cost estimation is uncertain

- *Cost estimation and effort planning:* The risks of inappropriate project schedules and underestimated budgets is high in conventional projects and higher in offshoring projects. Established methods for cost estimation and effort planning of offshoring projects are not available. The well-known approaches assume work in collocated places. They need to be adapted, refined and extended for onshore and offshore distribution of work.

Process quality

- *Quality of development process:* An appropriate, well-defined development process is necessary to be able to address problems and risks occurring in the development (e.g. incorrect requirements, lack of domain knowledge, design flaws, technological problems). Both partners have to adhere to this process. The risk of miscommunication is high if the onshore and offshore teams use different sets of methods and tools, or if they follow different process templates. Even worse is the case that the partners are at different levels of process maturity (e.g. different CMMI levels).
- *Project management:* Offshoring projects are more difficult to manage than onsite projects. Additional factors such as communication, coordination and management across countries and cultures need to be considered.

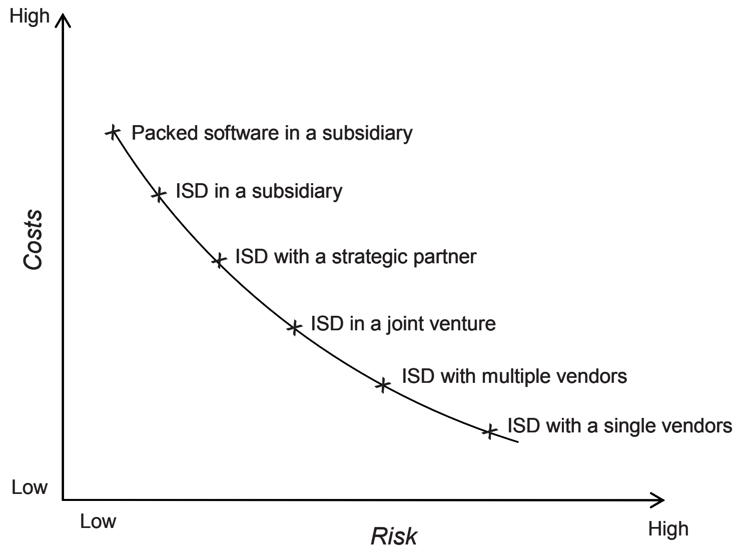
More detailed discussions of the risks in offshore software development are provided in the offshoring literature [e.g. Aspray 2006, pp. 182-212].

The *level of risk* in an offshoring project depends on the type of system to be developed and on the organization of offshoring [Sakthivel 2007, pp. 72-75]. The spectrum of systems with different risk levels shows strategic information systems on the one end and routine systems on the other end, with several variants in between. Strategic information systems involving new technologies, new business processes, and possibly evolving requirements bear high risks whereas routine systems with stable requirements have rather low risks.

Risks and costs depend on the organization of offshoring

The organizational form of offshoring as discussed in the beginning of section 2.3 can also be associated with higher or lower risks. This implies a trade-off between costs and risks (cf. figure 2-7). Setting up a captive center (subsidiary) exhibits significantly lower risks for an organization than working with a single offshore vendor and having to depend solely on them. However, the costs of setting up a captive center and communication infrastructure are much higher than the costs of finding and collaborating with a vendor who already has the necessary infrastructure. Figure 2-7 shows that the risk level and the costs, on the spectrum between these two extreme forms, are inversely proportional.

Figure 2-7 **Offshoring risks and costs**§



§ Adapted from: Sakthivel 2007, p. 73.

2.3.4 The Costs of Offshore Outsourcing

Obvious and hidden costs

Not all costs of an offshoring project are immediately visible. Obviously the cost of a deal agreed upon with the offshore-services provider, whether fixed-rate or hourly, is known or can be estimated, but cost factors such as knowledge transfer and transition are easily overlooked. Such hidden costs show when a detailed analysis of the project is performed. In this section we will take a closer look at obvious and hidden costs. Since more empirical data are available for offshore outsourcing and hardly any for offshoring to a captive center, we illustrate the costs to be considered by focusing on offshore outsourcing.

TCP project assessment

Cost factors can be distinguished in different ways. Siemens AG, among other things one of the largest software development organizations in the world, created a method for assessing offshore project candidates with the help of a comprehensive list of cost factors. This method is called TCP because it analyzes projects from a technical, a commercial (or business), and a process-related perspective. It is a general method for project assessment, not a specific method for information systems development. However, many of the considered cost factors occur in ISD projects as well as in other types of projects. Furthermore, TCP is focused on working with external partners and not on giving work to subsidiaries offshore (captive centers).

Cost factors are divided into one-off and recurring costs and differentiated according to technical, business, and process perspectives, as shown in figure 2-8 [Amberg 2005]:

Costs from a technical perspective

- *Task or process selection* – finding the right ISD task or the right business process for offshoring, from a technical point of view (e.g. required technical skills).
- *Knowledge transfer* – company knowledge required for the development has to be transferred to the offshoring provider, e.g. by training offshore personnel either at the customer's site (onsite) or at the provider's site (offshore), or by sending company experts offshore to work with the provider for the duration of the project.

- *Project specifications* – required not only in the beginning but also in every step of the process model. Creating specifications may be costly for complex systems due to integrity requirements and the level of detail.

Costs from a business perspective

- *Provider selection* – costs of preselecting possible outsourcing providers based on market data, narrowing down the list by evaluating detailed information, and onsite auditing of the remaining candidates. This process has been reported to cost an additional 1 to 10 % of the annual cost of an offshoring deal, taking from six months to a year [Overby 2003].

Figure 2-8 **Cost factors of IT offshore outsourcing [Amberg 2005]**

	Technical perspective	Business perspective	Process perspective
Non-recurring costs	Task or process selection Knowhow transfer	Outsourcing provider selection Contract management	Process synchronization Transition
Recurring costs	Project specification	Labor Risk management	Cooperation Perfomance measurement

- *Contract management* – costs of inviting bids, evaluation of quotations, contract negotiations, setting up and concluding the contract; monitoring adherence with the contract, change-request management, conflict management, invoicing, charging cost centers etc. in the course of the project. Additional costs range from 6 to 10 percent per year [Overby 2003].
- *Labor* – since software development is a labor-intense process, the cost of labor is the major cost portion. In outsourcing projects it is usually included in the contracted totals, but in captive offshore centers the labor cost is likely to be explicitly calculated, and made visible to the mother company. A loss in productivity may have to be considered in the beginning, because offshore personnel have to get acquainted with the task and processes, due to cultural differences, and for similar reasons.

- *Risk management* – costs of analyzing and evaluating risk factors, monitoring and keeping track of risks, developing and applying measures to minimize or avoid risks, etc.

Costs from a process perspective

- *Process synchronization* – costs of synchronizing the processes of the project partners, regular process auditing, reporting mechanisms, processes for early feedback, technical audits at the supplier site and common development guidelines.
- *Transition* – costs of what is required in order to hand the work over to the offshoring provider. This includes onsite visits to familiarize offshore developers with the processes, technology and architecture of the customer before these developers can begin the actual work in their home country. An adequate IT infrastructure with specific software and hardware and broadband data communication may need to be set up at the offshore site.
- *Cooperation* – costs of sustaining cooperation through meetings, traveling, communication, trouble shooting etc
- *Performance measurement* – costs of defining adequate performance metrics and monitoring the provider's performance in the course of the project.

Figure 2-9 **Additional offshore outsourcing costs [Overby 2003]**

Cost factors	*Ranges*		
Selecting a vendor	0.2	-	2%
Transition	2	-	3%
Laying off employees, severance, retention	3	-	5%
Lost productivity & cultural issues	3	-	27%
Improving development processes	1	-	10%
Managing the contract	6	-	10%
Total hidden costs			
Best case	15.2%		
Worst case			57%

Hidden costs of offshore outsourcing

Cost figures from practical experiences in large projects are illustrated by Overby in an article on the hidden costs of offshore outsourcing [Overby 2003]. It is assumed that offshoring is a long-term activity, so the annual cost of an offshoring deal as agreed upon with the contractor are known and can be used as a reference. Additional costs are estimated as a percent of the annual contracted cost in the ranges shown in figure 2-9. In the best case, these costs are 15.2 %. In the worst case, they add up to 57 %.

A sample computation based on these ranges is given in figure 2-10. It is assumed that the company's total value of offshore outsourcing contracts is $16.2 million per year (this happens to be the average value of offshore outsourcing contracts determined in a survey of 101 companies quoted by the author).

Figure 2-10 **Total cost of offshore outsourcing [Overby 2003]**

Hidden Costs	**Best Case**			**Worst Case**		
	Contract value			Contract value		
1. Vendor selection	$ 16.2 M	x .002 =	$ 32.4 K	$ 16.2 M	x .02 =	$ 324 K
2. Transitioning the work		x .02 =	$ 324 K		x .03 =	$ 486 K
3. Layoffs and retention		x .03 =	$ 486 K		x .05 =	$ 810 K
4. Lost productivity/cultural issues		x .03 =	$ 486 K		x .27 =	$ 4.4 M
5. Improving development processes		x .01 =	$ 162 K		x .10 =	$ 1.6 M
6. Managing the contract	↓	x .06 =	$ 912 K	↓	x .10 =	$ 1.6 M
Total hidden costs		15.2 % =	$ 2.5 M		57 % =	$ 9.2 M
Original contract value		+	$ 16.2 M		+	$ 16.2 M
Total cost of outsourcing (TCO)		**Best case =**	$ 18.7 M		**Worst case =**	$ 25.4 M

The costs of vendor selection are annualized in the table over 5 years from an initial cost of 1 - 10 %. The large span in the cost category "lost productivity/cultural issues" is caused by widely varying factors such as the maturity of the offshore provider, understanding of cultural differences among onshore and offshore workers, the turnover rate among offshore workers, and the length of the contract.

2.3.5 Special Issues of ISD Offshoring

Offshoring is not only practiced in software development but across business functions and processes. Infrastructure services, operative business processes, research and development, and many more activities are being performed offshore.

Offshoring has many variants

Many articles and reports discuss procedures, benefits, disadvantages and pitfalls of offshoring in a rather general manner, not distinguishing between different domains of work. However, offshoring a call center or flight-reservations processing is obviously quite different from offshoring information systems development. Even within the IT field, offshoring can mean quite different things. It makes a difference whether operations (e.g. processing bank transactions), a function such as running a computing center, or processes such as maintaining legacy systems and developing information systems are outsourced.

Offshoring study by an ACM task force

An excellent exception to the bulk of literature treating everything alike is the above mentioned offshoring study prepared by an ACM task force [Aspray 2006]. This study focuses clearly on offshoring software-related activities, and it makes clear differentiations between different types of work sent offshore.

Offshoring just one project?

Since the focus of this book is the making of information systems, we will point out some special characteristics of offshoring which are related to information systems development. ISD is usually done in the form of projects. A project is by definiton a unique undertaking and will not re-occur in the same form again. So an immediate question is: Should the offshoring cover only this one project? This would be a rare case, contrary to observed business practices. As we discussed earlier, offshoring is more a business strategy than a one-time activity, requiring a relatively long phase of preparation. Going through this just for one project might not be worth the effort.

The work of setting up an ISD offshoring project depends on the complexity of the problems to be solved and on the level of abstraction. The closer the outsourced work is to the coding stage, the lesser the effort is to get the project on track, because code is less vague than requirements or a statement of the business problem.

Outsourcing in projects

Outsourcing starts with ...

Looking at things bottom up, we can identify generic levels of difficulty in offshoring information systems development. Figure 2-11 summarizes these levels.

Coding

1. Outsourcing the coding and testing of a system from the domestic organization to an offshore organization is relatively simple, provided that the client delivers clear and precise system/module specifications. Unfortunately this simple precondition is not so simple to meet. Specifications are often ambiguous and incomplete, leaving room for (mis-) interpretation by offshore developers. Nevertheless, a coding project is easier to handle than the scenarios mentioned subsequently.

Figure 2-11 **Scope of ISD offshore outsourcing**

Outsourced activities	Input by customer	Output by offshore provider	Difficulty of project
Coding & testing	Module specifications, system specification, test cases	Running information system	↓
Module design, coding & testing	System specification & design divided into modules	Module specifications, implemented & tested modules, running information system	↓
System design, module design, coding & testing	Requirements specification	System specification & design, module specifications, implemented & tested modules, running information system	↓
Application problem	Project & problem scope, problem description	Requirements specification, System specification & design, module specifications, implemented & tested modules, running information system	↓

Module design

2. Outsourcing module design, coding and testing means that the client provides an architectural specification of the entire system and of what modules are expected to be there. Detailed module specifications are prepared by the offshore company and discussed with the client. Based on these specifications the system is programmed and tested.

System design

3. Outsourcing system design, module design, coding, and testing can be considered if the client's requirements are clear and well-specified. That is, requirements engineering is performed by the client. The offshore organization starts from the given requirements specification, developing an architecture if an architecture is not provided by the client. However, if the client creates a requirements specification only on paper, without a working system prototype, there is a high risk that this specification will not be correctly understood. Paper specifications are often ambiguous, imprecise and incomplete, increasing the need for extensive communication between onsite and offshore personnel.

Requirements

4. "Outsourcing the problem" means that responsibility for all activities related to the development of an information system is given to the offshore organization. Provided that the client has defined the project scope and decided that the information system will be built, the offshore organization starts by elaborating requirements and creating the requirements specification, followed by design, coding and testing. This is obviously the most challenging outsourcing situation with regard to communication requirements between the client's staff and offshore personnel.

Offshoring strategy

Looking at the offshoring of information systems development as a long-term business strategy, general policies and regulations beyond a particular project are needed.

Where to draw the line is a strategic decision

The enterprise's management has to decide where to draw the line. What will remain onsite, and what will go offshore? More precisely, which activities (or stages in the ISD process model) will be outsourced to an offshore location? Should they be outsourced completely, or will equal activities continue to be performed inside the company? For example, will all coding be outsourced in the future, or will onsite programmers continue to develop code as well? If onsite and offsite personnel work on the same system, integration becomes an even more challenging issue.

The management of offshoring projects will be revisited in chapter 8 of this book which focuses on project management issues.

2.4 The Business Value of IS: Costs and Benefits

Many IT investments in the past have been justified by the alleged strategic implications of information systems that give organizations a competitive advantage. More often, operational benefits such as faster workflows and cost savings due to improved work efficiency have stimulated ISD projects. On the other hand, investments in new information systems are under pressure because they usually cost large amounts of money.

The fundamental question asked by top management – are the costs justified by the benefits? – has to be answered before a new project will be given the green light. The same question may be asked again later in the project if the need arises to redefine the project or to decide whether to cancel or continue with the project.

In this section, we will discuss the benefits and costs of information systems and the methods to evaluate the benefits and the costs.

2.4.1 Benefits from Information Systems

Benefits from information systems can be classified into tangible and intangible benefits. Tangible benefits can be quantified and measured. Intangible benefits cannot be quantified immediately, however, they may lead to long-term quantifiable advantages.

Tangible benefits

Tangible benefits, for example, are higher sales figures. Consider a new information system capable of predicting customer demand in a certain region a longer time ahead and more precisely than before. Due to the system's predictions, production can then be adjusted faster, stores can be supplied with the appropriate quantities and therefore sell more, and storage costs for shelf warmers will be reduced. All these factors can be measured and assessed in monetary terms. Many tangible benefits are actually cost savings as shown in figure 2-12.

Strategic advantage and other intangible benefits

The problem with benefits from information systems is that the most interesting ones, those interesting for top management, are mostly *intangible*. For example, the strategic advantage which can be obtained with the help of an information system will eventually be seen when the organization reaches a larger market share or a higher level of customer satisfaction. This advantage is difficult to assess in advance because it is not certain to what extent the competitive advantage will be reached or that it will be reached at all. To some extent it depends on what the competition does. Maybe they are developing a similar or a better information system. On the other hand, the organization might suffer from a severe competitive disadvantage if it does not have such a system in the future but the major competitor does.

Figure 2-12 **Benefits from information systems [Laudon 2007, p. 566]**

Tangible Benefits (Cost Savings)
Increased productivity Lower operational costs Reduced workforce Lower computer expenses Lower outside vendor costs Lower clerical and professional costs Reduced rate of growth in expenses Reduced facility costs
Intangible Benefits
Improved asset utilization Improved resource control Improved organizational planning Increased organizational flexibility More timely information More information Increased organizational learning Legal requirements attained Enhanced employee goodwill Increased job satisfaction Improved decision making Improved operations Higher client satisfaction Better corporate image

Although the thesis that IS provides organizations with competitive advantages was seriously challenged by Carr in his article "IT doesn't matter", as we pointed out in section 1.1, there are many examples of strategic benefits that have been reached with the help of information systems.

Value-adding business concepts require powerful information systems

Henning Kagermann, CEO of SAP AG, and Hubert Österle, director of the Institute of Information Management at the University of St. Gallen (Switzerland), wrote a book on business models that is full of such examples [Kagermann 2006]. The authors express clearly that the true benefits of information systems come from value-adding business concepts and business models and not from information systems as such.

However, information systems are the means through which value-adding concepts can be implemented. Many of today's innovative business models could not be realized without powerful information systems. The benefits expected from new business models often depend on the availability of information systems supporting the models.

2.4.2 The Cost of Making Information Systems

To know how much an information system will cost is important not only for the decision to develop or order one but also for project planning and control. The cost of *buying* an information system seems to be obvious. In this case, at least one major cost item, the licensing cost, is known as a hard fact. Unfortunately, this cost is only one part of the total cost.

Early cost estimates are difficult

The cost of *developing* an information system is much more difficult to predict because this cost depends primarily on the effort that the development process will require. In a project, the earlier this cost has to be calculated, the less exact the estimate will be. Or vice versa: The later this cost is calculated, the better the understanding is of what needs to be done, how much time it will take, and how much it will cost.

In this section, the major cost factors of making information systems are identified. Methods to predict the costs will be discussed in the subsequent section. For the analysis of cost factors, we differentiate according to the various ways in which an IS can be obtained (explained in section 2.2.3).

1. Inhouse development

Software developers and more

The major cost factor of developing an information system inhouse is human labor, in particular software development personnel. However, this is not all. A closer look reveals a list of cost factors:

- Software development staff (e.g. programmers, systems analysts, requirements engineers, software/IS architects, testers)
- Project management staff
- Support staff (e.g. secretaries, accountants, technicians, cleaners)
- Buying/licensing and installing IT infrastructure if not available (workplaces, networking, communication devices; development tools, e.g. an integrated development environment – IDE) and utilizing that infrastructure
- Providing office space, heating, electricity etc.
- Traveling, meetings, communication
- Training software developers in new software technologies and tools
- Training users in the functionalities and handling of the IS before and after the system is installed
- Lost productivity of non-development personnel (e.g. from interviews and discussions with end-users in requirements engineering)
- Implementation and conversion

A rough estimate is that the overhead on top of the core software development personnel amounts to about the same as the total salaries of those developers [Sommerville 2007, p. 614]. If a software developer is paid €12,000 per month and 5 developers work on the same project, the total cost of the project will add up to €120,000 per month.

2. Developing with external partners

External software and collaboration costs

In addition to the costs mentioned above, costs for those tasks or parts of the system that are provided by external partners have to be taken into account. Additional effort from the organization demanding an information system is required because it needs to specify operational interfaces with external partners, communicate with those partners, and integrate externally developed components into the overall information system.

If the external partner resides in a different country, then both the obvious and hidden costs of offshore outsourcing as discussed in section 2.3.4 (e.g. vendor selection and transition) have to be included.

3. Ordering an individual turnkey solution

Main cost: contracted price

This case is operationally simpler for the customer since the main cost is the contracted price of the new information system. This cost is determined in a process starting with an invitation for bids and ending with awarding the contract to the selected vendor. Additional costs include:

– Traveling, meetings, communication
– Lost productivity of non-development personnel
– Training of users before and after installation of the system
– Implementation and conversion
– Necessary hardware and networking equipment if that equipment was not included in the turnkey solution.

4. Buying, customizing and extending standard software

Total cost of ownership (TCO)

In the past, many organizations opting for standard software were misled by the hope that the cost would only be the license cost of the software. This erroneous assumption lead to the adoption of the *total cost of ownership (TCO)* concept for software. In general, the total cost of ownership of an asset is considered to be the purchase price plus the additional costs of operation. For standard software, TCO components include the following cost factors:

– Software license according to the contracted license model
– Customizing the software (either with the help of the package vendor, specialized consultants or inhouse staff)
– Extending the software if important features are missing
– Integrating the standard package with the rest of the company's information systems
– Installation of the system on the company's hardware, networks and system software
– User training before and after installation of the system
– Implementation and conversion (often with the help of external consultants)

– Hardware, software and network upgrades depending on the requirements of the new information system

License costs are only 20 - 30 % of the TCO

It should be noted that in many practice projects the software license costs account for not more than 20 - 30 % of the TCO. The full cost of employing standard software packages is three to five times higher! Additional costs occur for customizing, extending and integrating the standard package. These activities may call for a project of their own, with cost factors as in points 1. to 3. above.

5. Employing an application service provider (ASP)

Cost factors are fairly simple to identify when an application service provider is employed. In this case the major cost factor is the price paid to the provider, according to the agreed payment scheme. However, there are other costs inside the organization to consider as in the above cases, in particular the costs of training users, implementing new procedures in the organization related to the ASP's software, and conversion from the old processes to the new ones.

2.4.3 Cost Estimation Methods

Since reliable cost figures are extremely important both for the decision to set up a project and for the allocation and control of the project budget, a significant number of methods to estimate that cost was developed in the past.

Estimating the effort needed to achieve the project result

Although there is a wide variety of methods, the common goal of most methods is to estimate the effort or the time it takes to achieve the result of the project, i.e. a functioning, implemented and running information system. Conventional estimation methods are aimed at a situation where an organization is doing the development itself (i.e. case 1. above).

Assumption: effort is proportional to system size

An assumption underlying many methods is that the effort required for the development (measured in person months) is proportional to the size of the future information system. If a size estimate is available, this number can be multiplied by a cost coefficient or used in an estimation function to yield the total cost of the development project.

An immediate question follows from this: How to measure the size of an information system? Common approaches are:

Lines of code (LOC)

- *Lines of code:* The traditional unit of measurement for software has been the lines of code (LOC) of the proposed system. Usually source-code lines are used, but machine-code instructions are used as well. Many authors and practicioners have argued that LOC is a questionable measure. A line of code in an assembler language is not comparable with a line of code in a third or fourth-generation language. Therefore projects can only be directly compared when they use the same language, and cross-language figures are meaningless. With the emergence of CASE tools, the lines of code became increasingly irrelevant because those tools often generate an abundance of code lines – much more than a human programmer would write. Nevertheless the lines of code have been the most used measure of system size.

Function points

- *Function points:* Due to the weakness of the lines-of-code measure, IBM introduced function points as an alternative measure in 1979. This measure is based on the system's functionality and not on its low-level implementation (code). Function points are given for program functions as the programmer sees them, e.g. input, output, data retrieval, external interfaces and related functions.

Object points

- *Object points:* A higher abstraction level than function points is assumed when system objects are used as a measure of size. Objects in this approach are not identical with objects in object-oriented programming. The term stands for screens, reports and code modules that have to be developed. These are typical programming objects when a fourth-generation language is used.

Most cost estimation methods that are based on the predicted size of an information system use one of these three measures of size.

Estimation based on size measures is not appropriate in early project stages

A common drawback of LOC, function and object points, and of estimation methods employing these measures is that they require a fairly detailed understanding of the design and the module structure of the future information system. Such understanding is hardly available before the project has started and requirements, architecture, interfaces have been specified etc. Obviously estimation methods using size measures are not appropriate for the time before the project starts but only for later stages. In the course of the project they can provide valuable information for project management, controlling and budgeting.

Nevertheless, cost figures are indispensable for the management decision to set up a project, but where do they come from? Other approaches that do not employ the system size and require less detail are available, but they provide results that are less precise.

Figure 2-13 **Characteristics of cost estimation methods**

Method	Description
Analogy-based	Previously completed similar projects are selected and compared with the current project. Actual data from the completed projects are used to estimate the requirements, duration, size etc. of the new project. Conclusions for the cost of the new project are drawn by analogy.
Case-based reasoning (CBR)	CBR is an automated artificial-intelligence approach using similarity and analogy. Information and knowledge about completed projects are stored with descriptors in a case knowledge base. A new project is checked against old projects for similarity. Costs and other properties of the closest old project are adapted to the characteristics of the new project.
Expert judgement	Experts on software cost estimation and/or the application domain and/or the required software technology are consulted. Those experts use their experience and understanding of the new project to arrive at a cost estimate. Several iterations or Delphi techniques may be applied to reach a consensus.
Percentage shares	This is a model used primarily to divide up the budget and/or human resource allocation according to project stages. Shares for the stages can be determined based on experiences from previous projects of the organization, or based on industry figures§.
Fixed budget	The project budget is set autonomously, for example based on what the top management provides for the project or what the customer is willing to pay. The estimated cost is not based on a detailed examination of the required system functionality but on what the budget allows.
Function point method	This is a size-based method which uses function points to estimate the size of the system, qualitative factors to include the levels of difficulty and complexity into account, and historical data (function-point curve) plotting function points against effort in person months.
Cocomo II	The Constructive Cost Model (Cocomo) is based on empirical data from many software development projects. Cocomo II provides formulae for different types of application systems that are used to estimate the effort for the new system in person months.

Typical approaches to predict the costs of developing an information system include the following:

§ A historical note: A rule-of-thumb often applied in conventional projects was the "40-20-40 rule", meaning that 40 % of the total effort goes into the early project stages up to system specification, 20 % goes into programming, and 40 % goes into testing, implementation and conversion.

- Analogy-based methods
- Case-based reasoning (CBR)
- Expert judgement
- Percentage shares
- Fixed budget

Combined approaches

It is worth mentioning that combinations of different approaches are often used. For example, a fixed budget is stipulated by the management based on analogies of former projects and/or expert judgement. This budget is then divided into portions for major project phases. Consequently, the size and functionality of the future information system is not only determined by market or user requirements, but also restricted by the given budget.

Figure 2-13 summarizes the main characteristics of a number of cost estimation methods. Apart from case-based reasoning (CBR), these are the most widely used approaches in practice. Despite this, CBR is an interesting method of artificial intelligence (AI) [Kurbel 1992]. The function-point method and the Cocomo model are discussed subsequently.

Function-point method

Effort depends on amount and difficulty of functions, and on system complexity

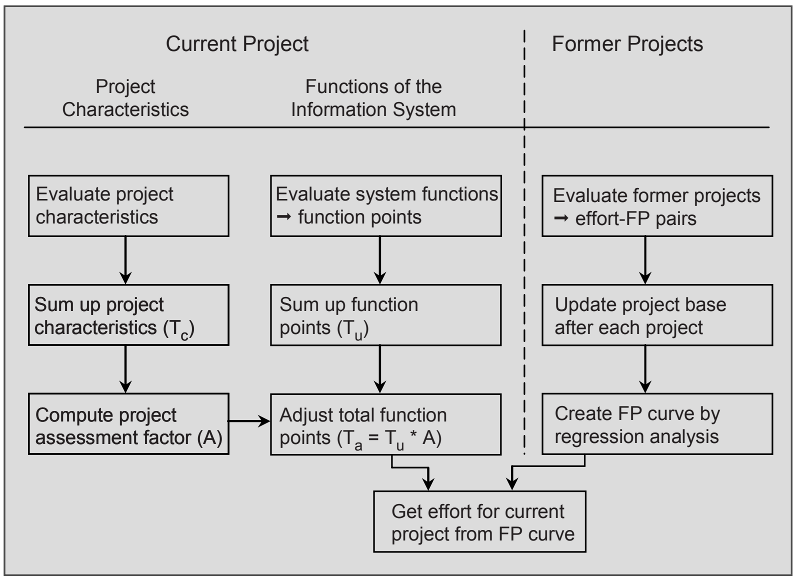
The function-point (FP) method was originally developed within IBM in 1979 [Albrecht 1983]. It is one of the few actual "methods" for software cost estimation that has been accepted and used by many organizations. This method relies on the assumption that the effort to develop an information system depends primarily on three factors: the functions of the future system, the difficulty of those functions, and the complexity of the project. The system's functions are evaluated, weighted, and the resulting points are summed up. The total number of function points is used to obtain the estimated effort from an empirical curve which is based on experiences from former projects. Figure 2-14 illustrates the major components of the method.

Programming-level functions

"Functions" in the FP terminology are functions on the programming level that were typical for early third-generation languages. They are assigned to the following categories:

1. Input functions (dialog input, batch input etc.)
2. Output functions (screens, forms, reports etc.)
3. Inquiries (user interactions requiring a response)
4. File manipulation
5. Interfacing other systems

Figure 2-14 **Components of function-point method**


Current Project
Former Projects
Project Characteristics
Functions of the Information System
Evaluate project characteristics
Evaluate system functions → function points
Evaluate former projects → effort-FP pairs
Sum up project characteristics (T_c)
Sum up function points (T_u)
Update project base after each project
Compute project assessment factor (A)
Adjust total function points ($T_a = T_u * A$)
Create FP curve by regression analysis
Get effort for current project from FP curve

System must be decomposed into functions first

The complete system has to be specified in terms of such functions. This is the first step, requiring a decomposition of the system into modules, programs and functions. Clearly such a decomposition cannot be done with sufficient accuracy before the system has been specified and designed in detail. This shows that the FP method is not an appropriate method for pre-project or initital project stages but for later stages. In an early project stage, at best rough function estimates can be used.

In the next step, each function is evaluated according to its level of difficulty. Weights for each type of function and each level of difficulty are predefined, based on previous experience. Let c_{ij} = weight of a function of category i and difficulty level j, and x_{ij} = function count for category i and difficulty level j, with $j \in \{simple, average, difficult\}$. The *function points* for category i are then obtained by adding the products $c_{ij} * x_{ij}$ for category i.

Example: a "difficult" and "simple" function

For example, a dialog-input function that checks the plausibility of user input by searching the system's database, provides context-sensitive user menus, and lets the user go back and forth between screens is considered *difficult*. A similar function that checks only whether the user input is numeric or not, provides only fixed standard menus, and allows no forward and backward navigation is considered *simple*. The difficult function will be weighted with 6, the simple one with 2 (and an average function, for example, would be weighted with 4). If there are 50 difficult, 20 average and 40 simple input functions in the prospective system, then the total function points for input functions are:

$$50*6 + 20*4 + 40*2 = 460$$

Function weights for the other types may have different ranges. For example, file-handling functions have often been considered more difficult, with ranges from 5 to 20.

The total number of function points T_u summed up over all functions of all categories is then:

$$T_u = \Sigma\Sigma\ c_{ij} * x_{ij}$$

Sum of function points weighted with the project's complexity

When the sum of the function points is computed in this way, this sum is weighted according to characteristics which reflect the complexity and the environment of the project. Such characteristics include the development platform, the degree of distributed processing, the amount of reuse, the required performance (e.g. response times), experience of the development team, degree of user involvement, and so on. The project characteristics are mapped to an assessment factor A that raises or lowers the initial total function points T_u in a range of $\pm$ 30 %.

More precisely, each characteristic is weighted with a value between 0 and 5. A value of 0 means that this characteristic has no influence in the current project whereas a value of 5 indicates very strong influence. The weighted characteristics are added up, yielding a value around 100.

When T_c is the sum of project characteristics, the project assessment factor A reflecting project characteristics is computed as

$$A = 0.7 + T_c /100$$

Total function points

Weighting the unadjusted function points total T_u with A finally results in the adjusted total T_a:

$$T_a = A * T_u$$

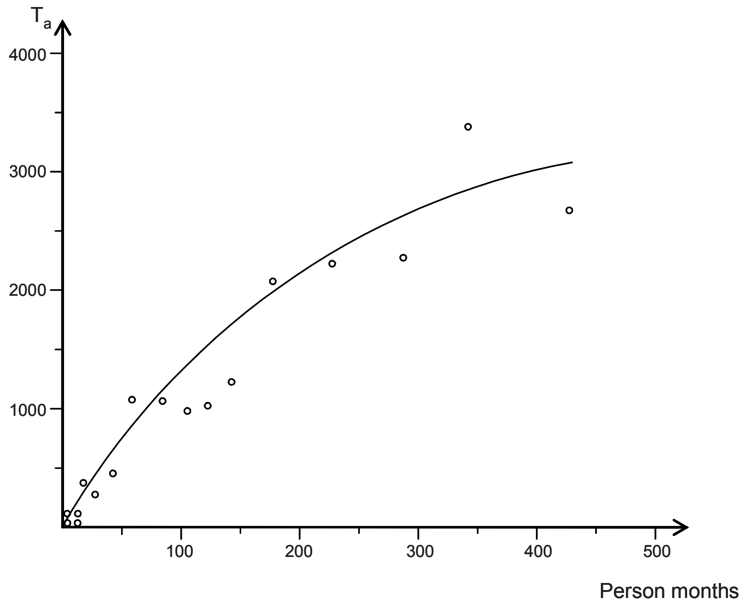
Function-point curve

Provided that the organization has collected experiences from earlier projects (i.e. assessed the projects according to the FP method), then the function points from those projects can be plotted against the effort needed in each project, yielding a set of points scattered in a two-dimensional space. By regression analysis a curve can be constructed through these points (FP curve) as in figure 2-15. The expected effort of the new project is then obtained by reading the effort (person months) corresponding to the T_a value of the new project from the FP curve.

Subjective factors

The function-point method has the advantage that it uses more objective measures than just gut level analogies or judgements. Nevertheless it involves subjective factors as well. Different people have different opinions of what makes a function simple or difficult, and of what makes a project complex and more/less problematic to handle.

Figure 2-15 **Function-point curve**



Modern IS are not function-oriented

Other drawbacks come from the fact that the FP method is rather old. It is a suitable method for data-processing systems with a significant share of input/output handling, designed in a function-oriented way, for example with an appoach such as SA/SD (Structured Analysis/Structured Design [DeMarco 1978]). Today's information systems have different characteristics, e.g. graphical user interfaces (GUI) that did not exist when the FP method was invented. They use database management systems instead of files, they are often event-driven, and usually they are designed as object-oriented systems.

International Function Point Users Group (IFPUG)

The FP method has been extended and refined to cover new system types and development approaches. The International Function Point Users Group (IFPUG), a non-profit organization whose mission is to promote the effective management of application software development through the use of function points, included rules for counting GUI based systems in their manual [IFPUG 2005]. The IFPUG claims that the FP method can be used for object-oriented systems as well and demonstrated this with a case study.

Cocomo II

Barry Boehm is the father of Cocomo

The best-known cost-estimation model today is Cocomo II. This is a very complex model, created by Barry Boehm and his research group at USC-CSE, the University of Southern California's Center for Software Engineering. Its predecessor, called Cocomo (Constructive Cost Model), had been developed by Boehm during his time as Chief Scientist of the Defense Systems Group at TRW, a Californian consulting firm.

Cocomo II's predecessor was Cocomo 81

Cocomo was based on empirical project data from a large set of projects at TRW. The original model was first published by Boehm in his famous book "Software Engineering Economics" in 1981 [Boehm 1981]. Cocomo II is seen by the authors as a model that evolved from the original Cocomo, accounting for changes in software-engineering methodology and technology, rather than as a replacement or withdrawal of the earlier concepts. To distinguish the original model from the upgraded model, the former one is now referred to as Cocomo 81.

Cocomo 81 was a suitable model for large software systems built to specification according to a linear development model such as the waterfall model (cf. section 4.2.1) and implemented with third-generation languages. However, business systems, object-oriented software and new approaches that became popular in the late 1980s and the 1990s – such as prototyping, composing solutions from off-the-shelf components, evolutionary and incremental development etc. – did not fit Cocomo 81 well.

Cocomo II submodels

An upgraded version of the model, Cocomo II, was created by Boehm and his group and published in 1995 [Boehm 1995a]. Cocomo II addresses the previously mentioned topics. It provides three submodels targeted towards different types of systems and different stages of a project:

- *Application composition model:* This model is for systems that are developed with the help of tools connecting interoperable components. Those components are created, for example, with GUI builders, distributed-processing middleware or database managers, or selected from domain-specific packages.
- *Early design model:* This model can be used before the system's architecture and design have been completed, i.e. in early project stages.
- *Post-architecture model:* When the architecture is specified and more details are known, the cost of the system can be estimated on a fine-grained level with the help of the post-architecture model.

Application composition model

The application composition model is recommended for prototyping projects and for software that can be put together from existing components.

Object points are used as a size measure

This model employs *object points* as a measure of size rather than lines of code or function points. An initial object count is obtained from the estimated number of screens, reports and third-generation language modules. Each such object is weighted according to its classification as simple, medium or difficult. Finally the reuse rate and the developer's productivity are taken into account. Boehm et al. propose a seven step procedure to arrive at the expected effort in person months [Boehm 1995b]:

Seven step procedure

1. Assess object counts, i.e. estimate the number of screens, reports and 3GL components that will constitute this application.

Classifying objects

2. Classify each object instance into simple, medium and difficult complexity levels depending on values of characteristic dimensions. What makes a screen simple, medium or difficult, for example, are the number of data tables where the data comes from, the distribution of those tables between servers and clients, and the number of views contained in the screen. For reports, the same criteria regarding sources of data are considered plus the number of sections the report has. 3GL modules are generally considered difficult.

Assigning weights

3. The weights associated with simple, medium and difficult objects as shown in figure 2-16 are then employed to reflect the relative effort required to implement an instance of that complexity level.

4. Determine object points: Add all the weighted object instances to get one number, the object-point count OP.

Determining OP count

Figure 2-16 **Object weights for application composition model**[§]

Object type	Complexity weight		
	simple	medium	difficult
Screen	1	2	3
Report	2	5	8
3 GL component			10

5. Estimate the percentage of screens, reports and 3GL modules expected to be reused in the project (%reuse). Compute the new object points NOP as:

   NOP = OP (100 - %reuse)/100.

Estimating reuse

6. Determine a productivity rate, PROD = NOP/person month, from the scheme given in figure 2-17. (The productivity rates in the figure were derived from an empirical analysis of project data.)

Determining productivity

Figure 2-17 Productivity in application composition model[#]

Developers' experience and capability	very low	low	nominal	high	very high
ICASE maturity and capability	very low	low	nominal	high	very high
PROD (NOP/person month)	4	7	13	25	50

§ Boehm 1995b.
# Boehm 1995b.

Computing person months

7. Compute the estimated effort for the project in person months (PM) as:

$$PM = NOP/PROD$$

Early design model

Early design model is used early in or before the design phase

As the name suggests, this model is used early in or before the design phase. At this stage of the system development, not enough is known for a fine-grained cost estimation. The early design model uses a relatively small set of cost drivers compared with the post-architecture model. Those cost drivers are the ones that can be reasonably assessed at an early point of development.

Function points and lines of code are used as a size measure

The early design model uses *unadjusted function points (T_u)* and *source lines of code (SLOC)* for estimating the size of the system. Functions points for the system under consideration are determined in a similar way as in the function-point method above. The unadjusted function points T_u are converted into source lines of code. Further computations are based on SLOC. Cocomo II provides detailed criteria of what is counted as a source line of code and what not.

Basic estimation equation

The basic estimation equation in the early design model yields the effort PM as:

$$PM = A * S^B * EA$$

where A is a constant calibrated with empirical data. It is proposed to be in the range between 2.5 and 3. S is the size of the system in KSLOC (kilo SLOC = thousand SLOC). EA is a multiplier for effort adjustment based on the cost drivers considered at this level.

Scale factors

B is an exponential factor that reflects increased or decreased effort as the size of the project increases (economies or diseconomies of scale). If $B < 1$, the project exhibits economies of scale. If the system's size is doubled, the project effort is less than doubled. For small projects, fixed startup activities such as tailoring tools and setup of standards can be a source of economies of scale. *Scale factors* are:

– Precedentedness (how familiar is the project?)
– Development flexibility (rigorous vs. general goals)
– Architecture/risk resolution (well-specified interfaces, extent of risk analysis carried out)
– Team cohesion (difficult vs. seamless interaction in the team)
– Process maturity (according to capabilities maturity model – CMMI)

Each factor is rated with a weight W_j from 5 (very low) to 0 (very high). Boehm et al. recommend setting the exponent B as follows:

$$B = 1.01 + 0.01 \Sigma W_j$$

Cost drivers

The effort adjustment multiplier EA is computed as the product of the numerical values obtained for seven *cost drivers* that are considered in the early design model:

- RCPX (required software reliability, database size, system complexity, documentation)
- RUSE (additional effort for required reusability)
- PDIF (platform difficulty)
- PERS (personnel capability)
- PREX (personnel experience)
- FCIL (facilities, e.g software tools, multi-site development)
- SCED (required development schedule)

Effort adjustment multiplier

Each cost driver is weighted on a scale from 1 (very low) to 6 (very high). The product of the weighted cost drivers yields the *effort adjustment multiplier:*

$$EA = RCPX * RUSE * PDIF * PERS * PREX * FCIL * SCED$$

An estimation example

As an example, consider a development project with system size SLOC = 12,000, an exponential factor B = 1.1 and A = 2.95. If no cost driver has any upwards or downwards effect (i.e. all cost drivers are 1), then the basic estimated equation yields a total effort of:

$$PM = 2.95 * 12^{1.1} * 1 = 45.4 \text{ person months}$$

Suppose the product of the cost drivers EA is different from 1, e.g. 1.4, then the total effort of the project is calculated as:

$$PM = 2.95 * 12^{1.1} * 1.4 = 63.5 \text{ person months.}$$

Code generation and reuse

Generated code

Nowadays significant portions of code may be *generated* with automated tools (e.g. ICASE tools). The productivity in terms of SLOC is much higher than for manually created code. Therefore the effort required for generated code may be computed separately and added to the effort for manually written code. Let

PM_a = additional effort for using code generation,

PM	=	effort for manually generated code as in the basic stimation equation above
PM_m	=	total effort for such a mixed system,
ASLOC	=	number of automatically generated source code lines,
AT	=	percentage of the total system code generated automatically,
ATPROD	=	productivity level for this type of code creation.

The estimated effort for the automated part is computed as [Sommerville 2007, p. 629]:

$PM_a = (ASLOC * AT/100) / ATPROD$

Then the total effort taking manually and automatically produced parts of the system into account is:

$PM_m = PM + PM_a$

Reused code

Code reuse is another characteristic of modern software development. Cocomo II considers reused code in such a way that it computes an equivalent number of lines of new source code (ESLOC) based on the number of reusable lines of code in the components that have to be adapted (ASLOC). The estimated project effort is then based on the equivalent. The formula for ESLOC takes into account the effort required to understand the software, to make changes to the reused code and to make changes to the system to integrate the new code [Boehm 1995b]:

$ESLOC = ASLOC * ((AA + SU)/100 + 0.4 * DM + 0.3 * CM + 0.3 * IM)$

AA (assessment and assimilation), SU (software understanding), DM (percentage of design modification), CM (percentage of code modification) and IM (percentage of integration effort) are called increments and are rated on scales with different ranges [see Boehm 1995b for details].

Post-architecture model

More cost drivers and more details than in the early design model

The post-architecture model uses the same PM estimation equation as the early design model, but there are more cost drivers which are more detailed than in the former model.

The *code size* in this model is determined by estimating three components:

1. the total number of lines of new code to be developed,

2. the number of equivalent source lines of code (ESLOC) considering reuse,
3. the number of lines of code to be modified because of requirement changes,

and adding these components.

Cost drivers

The *cost drivers* are grouped into four categories: product factors reflecting characteristics of the new system, hardware-platform factors constraining the project, personnel factors taking experience and capabilities of the project workers into account, and project factors reflecting project characteristics such as the software technology used.

Figure 2-18 summarizes the cost drivers for the post-architecture model. Each cost driver is rated on a scale from very low to extra high. Detailed criteria for this rating are available. The weights determined in the rating are mostly numbers between 0.8 and 1.3 (some ranges have end points as low as 0.67 and as high as 1.67).

Calibrating a Cocomo II model Result is highly sensitive to multiplier values

Judging the cost drivers is not free from subjective elements. Sommerville demonstrates the effects of the multipliers by a small example in which the initial value of B is 1.17. The cost drivers RELY, CPLX, STOR, TOOL and SCED are considered with values ≠ 1, and the resulting development effort is *730* person months [Sommerville 2007, p. 634]. If the five cost drivers are set to their maximum values, the result is *2,306* person months. This is more than three times the initial estimate. If the minimum values are taken, then the effort is *295* person months or 40 % of the initial estimate.

This variation in the results highlights that the people responsible for cost estimation need thorough experience with the Cocomo II model to arrive at reasonable estimates. This experience cannot be easily transferred from one project type or application domain to another. Cocomo II requires many details that need to be elaborated and calibrated for each user organization separately. Sommerville's bottom line is that "... it is an extremely complex model to understand and use ... In practice, however, few organisations have collected enough data from past projects in a form that supports model calibration. ... for the majority of companies, the cost of calibrating and learning to use an algorithmic model such as the Cocomo model is so high that they are unlikely to introduce this approach" [Sommerville 2007, p. 634].

Additional Cocomo II features

Cocomo II has additional features supporting the calculation of hardware cost (target hardware), platform cost, manpower cost and the duration of the project.

The total duration of a project depends on many factors. Since a short time-to-market may give the company a competitive advantage, managers tend to demand short development times. Putting more personnel into a project does not necessarily mean that the project will be complet-ed faster. In particular, if a project is behind schedule, more people may cause more problems. "Adding manpower to a late project makes it later" is an often quoted phrase by Frederick Brooks, a software engi-neering pioneer [Brooks 1995, p. 25]. Although more staff does not al-ways mean slowing down the project, it is obvious that more people have to spend more time communicating and specifying their interfaces.

Figure 2-18 **Cost drivers in the post-architecture model [Boehm 1995a]**

Product factors	
RELY	Required system reliability
DATA	Size of database used
CPLX	Complexity of system modules
RUSE	Required percentage of reusable components
DOCU	Extent of documentation required
Platform factors	
TIME	Execution time constraint
STOR	Main memory constraints
PVOL	Volatility of development platform
Personnel factors	
ACAP	Capability of project analysts
PCAP	Capability of programmers
AEXP	Analyst experience in the application domain
PEXP	Platform experience
LTEX	Language and tool experience
PCON	Personnel continuity
Project factors	
TOOL	Use of software tools (weak, powerful, ...)
SCED	Required development schedule (tight, comfortable, ...)
SITE	Multisite operations (collocated, distributed, international, ...)

Duration estimate

The project's duration TDEV can be derived from the computed effort figure PM. The exponent in the estimation formula accounts for the diverse factors that may influence the elapsed time:

$$TDEV = 3 * PM^{(0.33 + 0.2 * (B-1.01))}$$

In case the project schedule has been compressed (or expanded) compared to the initial schedule, the percentage of the compression or expansion can be considered through a factor P_{SCED}:

$$TDEV = 3 * PM^{(0.33 + 0.2 * (B-1.01))} * P_{SCED}/100$$

To illustrate the computation of TDEV, consider the above development project with system size SLOC = 12,000, an exponential factor B = 1.1, A = 2.95, and EA = 1.4. PM = 63.5 person months was yielded by the estimation equation.

Assuming that the project schedule was compressed to 80 % (P_{SCED} = 80), the duration of the project is computed as:

$$TDEV = 3 * 63.5^{(0.33 + 0.2 * (1.1-1.01))} * 80/100 = 10.2 \text{ months.}$$

Cocomo II is a "tuned model"

It should be noted that Cocomo II is not an academic or scientific approach but based on observations and data from real projects. The values of cost drivers, weights, scale factors etc. have been calibrated and adjusted over the years. We might say that Cocomo II is a "tuned model" based on real-world observation rather than an analytic model.

2.4.4 Cost-benefit Analysis

Will a new information system be worth making? Managers like to base their decisions on financial figures. Spending money on an asset that will hopefully produce returns in the future is an investment. Translating the question into management terminology results in something like: will we earn money from investing in the making of the new system? Or in other words: what is the return on the investment if we buy or develop that system?

Capital-budgeting methods

In the field of *capital budgeting*, a variety of methods are available to assess the profitability of an investment. Common methods are:

» Payback period
» Accounting rate of investment (ROI)
» Cost-benefit ratio
» Net present value
» Profitability index
» Internal rate of return (IRR)

Cash inflows and outflows

These methods are based on cash flows associated with buying or creating an asset. Money for the investment is spent in the beginning (cash outflow). Benefits from the investment are obtained later in the form of net cash inflows. Capital-budgeting methods weigh the cash flows going out of the company against the cash flows coming into the company, yielding a measure of the profitabilty of the proposed investment.

The main problem with capital-budgeting methods is that costs and benefits have to be expressed in financial numbers. Our discussion in the previous sections showed that predicting the costs of a new information system is already difficult, yet it is still easier than grasping the benefits. Costs tend to be tangible, and the major part of the costs occurs in the near future. Benefits are often intangible, achieved later, and cannot be expressed directly in financial figures.

Therefore it is helpful to distinguish between two scenarios, depending on whether cash flows are certain or uncertain.

Scenario 1: Certainty regarding cash flows

Reliable estimates of cash inflows and outflows

Scenario 1 comprises IS projects where reliable estimates of cash inflows and outflows are available. Consider, for example, a retail chain selling fashion clothes totaling €120,000,000 per year in their stores in several countries. Sales are reported to the company's headquarters at the end of each month. The marketing department found that the company could sell 20 % more in the next four years if they had real-time sales data available. In this case, production, procurement and delivery could be adjusted quickly to respond to changing customer behavior. An appropriate information system would be able to collect, evaluate and aggregate real-time data and make the needed information available to the production and sales managers.

20 % of €120,000,000 is easy to calculate (€24,000,000). Assuming that the additional cost of operations is €19,000,000 in the first year and €15,000,000 per year afterwards, the net benefits are €5,000,000 in the first year and €9,000,000 in each of the following three years (cash inflows).

A suitable information system offered by a vendor of standard business software has been selected. The total cost of ownership amounts to €15,500,000 in the first year and €2,900,000 in the following years. In the TCO, new hardware, software and network components are included as well as support, maintenance and software licenses. Figure 2-19 summarizes the example data.

Figure 2-19 **Cash flows of an IS project (example)**

Cash flow \ Year	1	2	3	4
Costs	15,500,000	2.900,000	2,900,000	2,900,000
Benefits	5,000,000	9,000,000	9,000,000	9,000,000
Net cash flow	-10,500,000	6,100,000	6,100,000	6,100,000

Given the above data, a capital-budgeting method can be used to support the decision making. Since cash inflows that will occur in the future are not of the same value to the company as money that is available today, the cash flows have to be discounted.

Net present value (NPV)

A method that takes the time value of money into account is the *net present value (NPV)* method. The net present value of an investment is computed as the sum of the expenditure in the first period (negative value) plus the discounted cash flows from future periods. Let x_i be the cash flow in period i, p the interest rate, and n the number of periods in which cash flows will occur, then the net present value of the investment is:

$$NPV = \sum_{i=0}^{n} x_i / (1+p)^i$$

with x_0 = initial investment (or net expenses in the first period). An investment is considered favorable if the net present value is positive. If NPV < 0 the conclusion is that investing the money in the project will result in a loss. The investor would get a better return if the money was invested elsewhere at an interest rate of p %, for example buying bonds in the capital market with an effective yield of p %.

Assuming an interest rate of 5 % and applying the NPV formula to the above example, the net present value of the new system is:

$$NPV = 6,111,812.98$$

This NPV means that the company will earn an equivalent of €6,111,812.98 (today's value) from the new information system; i.e. the decision should be in favor of the system.

It should be noted, however, that this conclusion is only true if the future sales are really 20 % higher than at present (€24,000,000). If they grow only by 18 % (€ 21,600,000), the net benefits will be €2,400,000 lower than expected, i.e. €2,600,000 in the first year and €6,600,000 in the following years (assuming that operational costs remain the same). Now the net present value comes out negative, meaning that the investment will result in a loss:

NPV = -2,823,982.29

This example shows that results of the NPV method, just as results of other capital-budgeting methods, are only trustworthy if the underlying assumptions are satisfied, in particular that the cash flows can be predicted with certainty.

Scenario 2: Uncertainty regarding cash flows

Intangible benefits

Unfortunately the benefits of most information systems do not lend themselves easily to quantification and measurement in units of money. Rather they are intangible, for example improving the firm's planning and decision-making infrastructure, streamlining business processes, or opening up new business opportunities. Capital-budgeting methods do not help much in those cases.

The making of information systems is not so much different from other situations where management decisions are required but reliable figures are not available. Managers use qualitative judgement and experience, consider cause-effect relations, and weigh market opportunities against risks to arrive at a decision. Qualitative approaches such as scenario techniques, balance of arguments and cause-effect chains can be used to support decision making when crisp numbers are not in sight.

Cause-effect chains

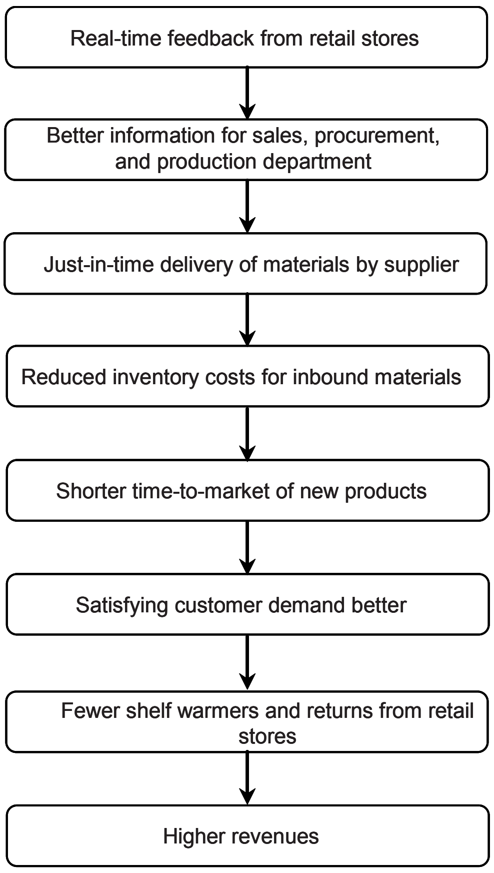
Cause-effect chains or networks, for example, put intangible benefits into a logical order, exhibiting implications of one benefit on other benefits. At the end there should be an effect that represents a financial goal or can easily be translated into such a goal. Figure 2-20 illustrates a cause-effect chain of benefits for a project similar to the one discussed in scenario 1.

If such a cause-effect chain is sufficiently convincing and an acceptable cost of the new information system can be predicted with sufficient accurateness, the decision will be in favor of the system.

As we pointed out before (see section 2.1), the forces stimulating new information systems are often market driven. When an IS might help to open new business opportunities or to satisfy important target groups demanding new services from the firm, qualitative benefits have a good chance of being convincing enough. The management decision to obtain such a system is likely to be based on "fuzzy" expectations rather than on precise cash-flow figures that are not available at this point anyway. In the absence of reliable figures, a typical decision situation is characterized by:

– a stated requirement from the market,
– a commitment to satisfy that requirement,
– the assignment of a project leader,

Figure 2-20 **Cause-effect chain of a proposed information system**


Real-time feedback from retail stores
Better information for sales, procurement, and production department
Just-in-time delivery of materials by supplier
Reduced inventory costs for inbound materials
Shorter time-to-market of new products
Satisfying customer demand better
Fewer shelf warmers and returns from retail stores
Higher revenues

– a project budget,
– a deadline when the system is expected to be operational.

Management decisions

When the decision to obtain the IS is made under uncertainty and with incomplete information, the budget and the deadline will only by chance be appropriate to meet the intended project goals. What happens when the project exceeds the budget or hits the deadline? Again a management decision will be made regarding a possible extension of the budget and/or the deadline, a reduction of the project goals or the cancellation of the project as discussed in section 2.2 above.

3 Information Systems Architecture

3.1 What is an Architecture?

"Architecture" is a popular term, yet different people use it for different things and with different meanings. In the 1980s and 1990s, architecture was close to becoming a buzzword. Apart from computer architecture, terms like enterprise architecture, information architecture, application architecture, communication architecture and more appeared to be trendy.

Different elements working smoothly together

The discussion in chapter 1 showed the need for quite a number of different elements to work together smoothly. Depending on the level of abstraction, such elements may be entire information systems such as a CRM system, web and application servers, database management systems, browsers etc. In a fine-grained view, elements may be programs or program modules, databases, data structures, classes, objects, user-interface forms or similar entities. Elements have to be arranged in a meaningful and effective way. The ease of adding new elements to the system and removing existing ones is important.

"Structure matters"

What exactly is an architecture? An architecture has very much to do with system structure. "Structure matters," is a key statement in a well-known book on software architecture [Bass 2003, p. 44]. An architecture defines the elements of a system, what they are meant to do, and their interrelations. Every non-trivial system has an architecture, whether it is implicit or explicit. A building has an architecture, a computer has an architecture, and software has an architecture. Booch calls a software architecture *intentional* if it has been explicitly identified and implemented, whereas "an accidental architecture emerges from the multitude of individual design decisions that occur during development [Booch 2006, p. 9]."

The study of software architecture as "... the principled understanding of the large-scale structures of software systems" [Shaw 2006, p. 31] emerged in the late 1980s. Since that time, intensive research in the field has made software architecture an essential part of system design and construction. An overview of the evolution of software architecture is given by Kruchten and coauthors [Kruchten 2006].

Enterprise-wide architecture vs. information system architecture

When discussing architecture, it is important to define the scope: Are we taking an organization-wide view, or are we talking about one information system? Hence a common distinction in the past was between an *enterprise-wide architecture* and an *information system's architecture* (sometimes called *software architecture*). While the latter is limited to the elements of just one system, the former represents a framework for all information systems in the organization.

Different architectures can coexist

At present we consider this distinction reasonable because different systems with different structures do coexist in reality. They all have their individual architectures: SAP ERP has its architecture, Microsoft Dynamics has its architecture, Siebel CRM has its architecture, etc. Any information system built around any of these systems must match the respective architecture.

Yet we believe that in the future the distinction between an enterprise-wide architecture and an information system's architecture will become obsolete. With the emergence of enterprise-wide software platforms, standard software vendors will place all their systems on such platforms. Likewise, user organizations will base individual new information systems on the same platform as the rest of their information systems. Using the same software infrastructure will have a standardizing effect on all information systems. In section 3.5, software platforms and their relationship with architecture will be discussed.

Enterprise architecture

Stressing an organization-wide integrative view of systems and business needs, the term *enterprise architecture* has been coined. An enterprise architecture describes how business processes, data, programs and technologies come together. Enterprise architects make all these parts fit together and fit into the governing principles of the enterprise [ASUG 2006, p. 9]. Enterprise architects take a holistic perspective.

For the above reason, we give just one generic definition of the term information systems architecture, extending the definition of software architecture by Bass et al. [Bass 2003]. The definition comprises the architecture of a single information system as well the architecture of an enterprise-wide set of IS.

Definition: information systems architecture

> An *information systems architecture* is the architecture of a usually large information system that may contain subsystems. Architecture refers to the structure or structures of the system, which comprise the elements of the system, the externally visible properties of those elements, and the relationships among them.

Referring to the externally visible properties of the elements implies that an architecture is an abstraction using the encapsulation principle [Parnas 1972a].

A system can have more than one structure

Since structure depends on the perspective of the viewer and on the type of relationship between the elements relevant for the viewer, a system can have more than one structure. Often one structure dominates, but others may be present. Note that "properties" is not being used here in the narrow object-oriented sense which describes only static attributes, but in a general sense which includes behavior. Thus the externally observable behavior of the elements is part of the architecture.

What makes an architecture a "good" architecture? Fundamental attributes of a quality architecture are:

– Robustness
– Stability
– Flexibility

An architecture is *robust* if structural changes can be performed without disturbing the entire architecture. *Stability* means that the architecture can survive for a significant period of time. A stable and robust architecture will allow for changes but basically remain the same over time. New versions of software products, for example, will not require the architecture to be redesigned.

Flexibility is an important attribute of a system's architecture

Architectural *flexibility* is a very important attribute today. In a dynamic world, software elements are changing rapidly. The architecture must allow the exchange of existing elements and the integration of new elements without major efforts. This calls for consistent application of the abstraction, information-hiding and encapsulation principles that go back all the way to the early 1970s [Parnas 1972b].

3.2 Common Architectural Patterns

In this section, we will discuss common architectural patterns, starting with a look at how the study of software architecture has emerged during the past decades.

3.2.1 Flashback to System Structures

"Structure" is a buzzword unless the relationship type is defined

Information hiding, hierarchy and layers of abstraction are the pillars on which architectures are built. Like information hiding, the concept of hierarchy was defined by Parnas in another famous article: "On a 'Buzzword': Hierarchical Structure" [Parnas 1974]. Parnas insisted that "structure" is a meaningful term only if there is a precise understanding of the *type of relationship* between the elements.

"Uses relation" (Parnas)

Parnas proposed the "uses relation" for that purpose. For a software system decomposed into modules, the uses relation is defined as follows: Module A uses module B if correct execution of B may be necessary for A to complete the task described in its specification [Parnas 1979, p. 131]. The uses relation was an appropriate relationship type for the development of individual software systems underlying the discussion at that time.

Abstraction layers

Abstraction layers in information systems development were discussed for the first time in the 1970s. A famous operating system developed by Edsger Dijkstra, the T.H.E. system, served as a model [Dijkstra 1968b]. The elements of that system were arranged in layers.

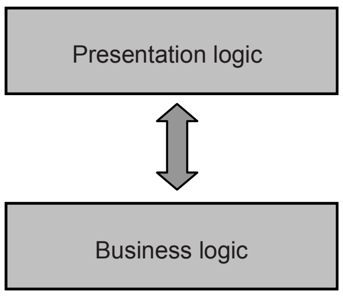
Layers were encapsulated following the concept of virtual machines. Lower layers provided abstractions of their respective functionalities to higher layers.

Transporting this concept from operating-system to application-system development was not straightforward and did not reach widespread practical use. However, the basic idea of a layered system structure returned and gained wide acceptance many years later when layered architectures emerged.

GUI and business-logic layers

The 1980s brought workstations and personal computers with graphical user interfaces (GUIs). These computers were not only used as stand-alone machines but also as front-ends to business information systems. Since GUIs need their own processing logic, this logic was isolated and assigned to a dedicated layer as in figure 3-1. One layer contains the graphical user interface and another layer contains the actual system logic.

Figure 3-1 **Separation of concerns in business information systems**



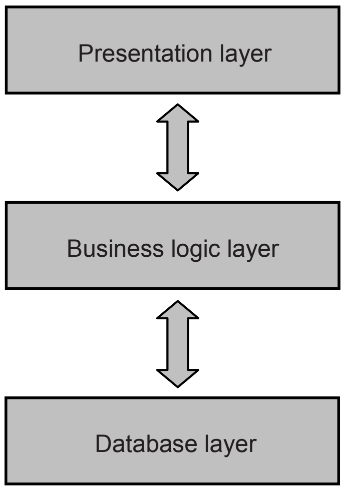
Three layers of a business information system

Information systems are usually built on databases. Preparing access to the database and accessing the database comprises a significant portion of an information system's code. Therefore a further division of labor appeared appropriate following the advances of the 1980s. GUI related tasks, database related tasks, and the actual logic of the underlying business problem were separated and assigned to different layers. According to this separation of tasks, a common pattern for business information systems contains three layers as shown in figure 3-2:

– Presentation layer
– Business logic or application layer
– Database layer

Since the tasks on the three layers have often been assigned to dedicated servers that can be accessed by clients, the term "client-server computing" was invented.

Figure 3-2 **Three-layered structure of a business information system**



Client-server model

The *client-server model* is a model for distributed computing that divides processing between clients and servers. Clients request services from servers, and servers provide those services. While the client-server model is actually a software model which can be implemented on any configuration of hardware, a common way is to assign servers and clients to separate computers. That is why many people associate hardware components with the terms servers and clients, and actually consider the client-server model a hardware model.

In the field of business software, SAP was one of the first companies to use this new model, introducing its R/3 enterprise resource planning system as a client-server system in 1990. Subsequently, most business information systems developed in the 1990s used the client-server model, applying the basic principles of forming layers and separating concerns into layers.

Modifications were now much easier than in monolithic systems, such as when introducing new GUI versions. Changing the application logic was also simplified as the overhead from user-interface code and database access no longer applied. That overhead often accounted for the largest share of an information system's code.

Thin and fat clients

In a client-server system, work can be divided up among clients and servers in many ways. If most tasks are assigned to the server(s) and little work is left for the clients, such clients are called *thin clients*. They do not require much computing power, so the client software may run on simple computers – at the end of the spectrum even on "dumb" terminals. In the opposite direction, if the clients perform a significant share of the work, they are called *fat clients*. Such clients obviously require more powerful computers.

Client-server systems with thin clients are easier to administer because most of the software is centralized on a few dedicated servers. Likewise, security hazards are easier to control on a server than on many clients. Fat clients are more convienent for the user because some tasks are executed directly at the user's computer, avoiding network traffic and slow responses.

With web-based front-ends for information systems and web browsers as the dominating user interface technology, clients have become rather thin. This development, however, created problems for systems requiring intensive user interaction beyond clicking on links, such as typical business information systems today.

Rich clients, AJAX, RCP

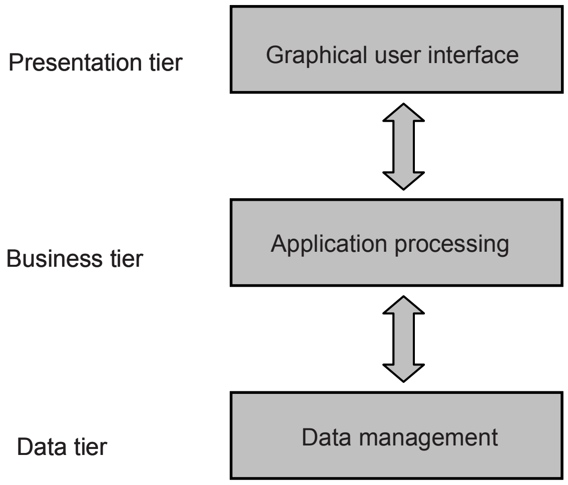
Nowadays a trend to bring more system functionality back to the client can be observed, making clients fatter again. *Rich client* is a term used for a client that provides more functionality than a simple browser-based client. On a rich client some of the processing can already be done, avoiding interaction with the server. Current technologies used for this purpose include AJAX (Asynchronous JavaScript and XML) and Eclipse RCP (Rich Client Platform).

3.2.2 Three-Tier and Multi-Tier Architectures

Three-tier architectures

As the idea of dividing systems into layers spread throughout the 1990s, the term architecture became popular. Layers were now called "tiers." A system structure as in figure 3-3 was called a three-tier architecture.

Figure 3-3 **Three-tier architecture**



Three-tier architectures became the dominating paradigm for many years, yet new requirements made further separations of tasks and layers necessary. In particular, electronic commerce and electronic business created new requirements, such as access to information systems via Internet and web browsers. In such a case, the presentation tier is actually represented by the browser, but now a web server and an application server had to be integrated into the architecture.

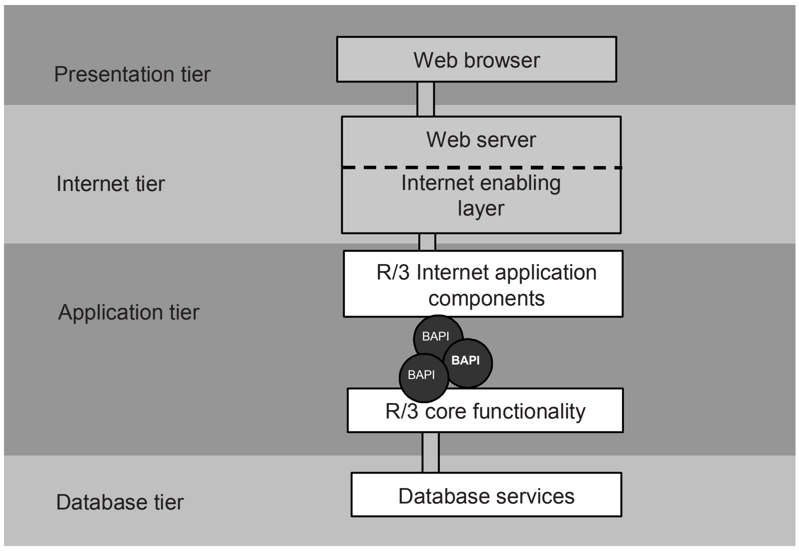
Multi-tier architectures

In this way, four-tier and multi-tier architectures came into existence. SAP, for example, introduced an architecture for an Internet-enabled R/3 release in 1997 that was composed of four tiers (or six, depending on the interpretation of tier division) as shown in figure 3-4. In addition to the three common tiers, an Internet tier was embedded. That tier contained a web server and functionality supporting Internet technology. The business or application tier was subdivided into R/3 core application functionality and Internet application functionality. These components communicate via SAP's BAPI (business application programming interface) mechanism.

End-devices in mobile business

Another reason to extend the three-tier model was the emergence of mobile commerce and mobile business, and the variety of end devices that employees and business partners use to access a firm's information systems.

Figure 3-4 **Internet enabled multi-tier SAP R/3 architecture [SAP 1997]**

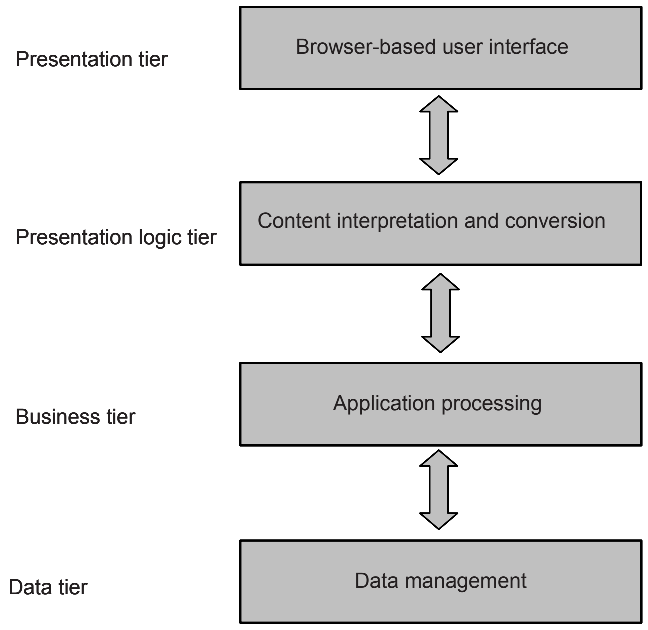


Examples are simple mobile phones with WAP and WML; more powerful ones with HTTP and XHTML MP or with Java ME; PDAs, palmtops and pocket PCs with XHTML or HTML etc. It is a long way from the firm's database via the business tier to the user's device-specific browser. Device-independent data has to be converted on the way from the business tier to the user and vice versa. In addition, data has to be adjusted to the properties of the respective end device, such as display size, graphics formats, available device memory etc.

Presentation logic tier

An architecture for mobile business systems is shown in figure 3-5. Compared with a conventional three-tier architecture, additional complexity is introduced by the tasks necessary to establish an appropriate user interface. These tasks are rather voluminous, involving interpretation and conversion of data in either direction. For this reason, a separate tier for the complex presentation logic is introduced, while the presentation as such remains on the top tier, realized by device-dependent browsers.

Figure 3-5 **Four-tier architecture for mobile information systems**



Tiers impose a clear static structure on a software system

Multi-tier architectures essentially impose a clear static structure onto a software system. Each system element belongs to a particular tier. The elements are interconnected according to one or more relationship types.

The overall structure of the system is static in the sense that the system is formed by the collection of all present elements. The logical view is that of one self-contained system, even though the system elements may reside at different physical locations, e.g. on different servers and clients connected by a network. (We could call such a system a "monolith", on a high abstraction level, if the term "monolithic system" had not been coined in the old times to describe non-modular systems.) The elements are static parts of the system, intended to remain what they are and where they are.

The perspective of conventional architectures such as a three-tier or multi-tier architecture is that a software system is composed of modules. Such modules may be procedures, forms, objects etc. – i.e., pieces of program code are regarded as elements of the system. A completely different view is taken when services (instead of software modules) are considered the constituents of an architecure.

3.3 Service-oriented Architecture (SOA)

Service-oriented architecture (SOA)

In a *service-oriented architecture (SOA)*, a system is regarded as a collection of services. The SOA perspective of architecture is thus on a higher abstraction level. Just as in real life, where people and businesses are increasingly interested in obtaining services instead of just products (objects), both the developer and the user of a software system will attach higher importance to getting the specified work done (i.e. obtaining a service) than to knowing which software module or modules are performing that work.

Services exchange messages

While software modules of a conventional system are invoked through method or procedure calls, a service exchanges *messages* with other services. That is, the interface of a service is constituted by the messages defined for communication with other services. Of course there must be software modules behind a service interface doing the requested work, yet these modules are completely hidden. Services exhibit strong information hiding.

Services may be obtained from anywhere in the world

In contrast to a conventional system, a system with a service-oriented architecture is not monolithic – neither physically nor logically. The opposite is true. Services may be obtained from anywhere. There is no need for the code implementing the service to be on a local server nor inside the organization at all. The service may be invoked via the Internet from anywhere in the world. The same service may be used in different information systems, ideally by different organizations independent of their geographic location.

Before proceeding further, the terms *service* and *service-oriented architecture (SOA)* have to be defined.

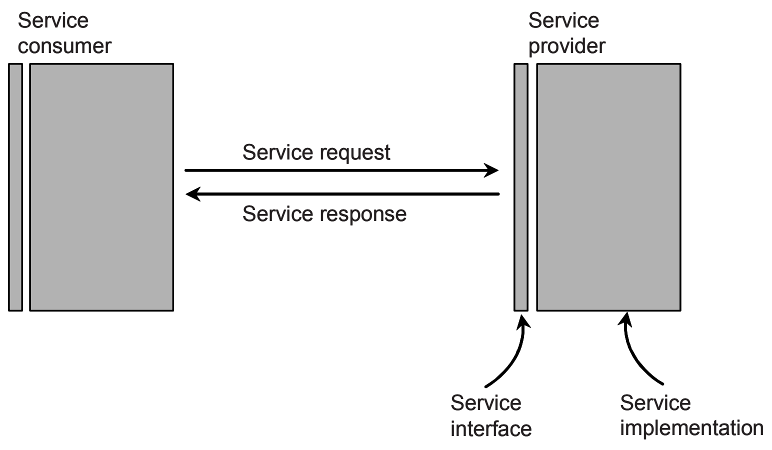
The notion of a software service is actually adopted from the notion of services in a business context. Customers or clients demand services from businesses, e.g. getting a quotation, booking a flight or opening a bank account. Likewise, a software service provides some functionality that is useful to software clients. A service provides a function that is well-defined, self-contained and does not depend on the context or state of other services [Barry 2003, p. 18]. A service accepts requests and returns responses through a well-defined, standard interface as illustrated by figure 3-6.

A formal definition of the term service was given by the W3 Consortium [W3C 2004]:

Definition: service

A *service* is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities. To be used, a service must be realized by a concrete provider agent.

Figure 3-6 **Service request and response**



A service-oriented architecture is essentially a collection of services that are capable of communicating with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity [Barry 2003, p. 18]. A more formal definition is as follows:

Definition: service-oriented architecture (SOA)

A *service-oriented architecture (SOA)* is a software architecture that defines the use of services to solve the tasks of a given software system. These services can be employed by other services in a standardized way. Services interoperate based on a

> formal definition which is independent from the underlying platform and programming language.

The services that constitute a particular architecture may be integrated with the help of technical infrastructure components such as a service bus and a service repository.

It may be noted that the term service-oriented architecture is often defined and used in a rather general way, not referring to an "architecture" in the actual sense, but calling SOA a "methodology" or a "design style" for interoperable systems, for example.

3.3.1 Web Services

The basic idea of a service-oriented architecture is independent of a particular software technology. However, the popularity of SOA at the beginning of the 21st century coincided with the emergence of web services as a new interoperation technology that is based on standard Internet protocols. The SOA paradigm as such is not an entirely new paradigm. It was already proposed earlier in the 1990s, but nowadays service-oriented architecture is often directly associated with web services.

Based on the W3 Consortium's definition [W3C 2004], we will use the term web service in the following sense:

Definition: web service

> A *web service* is a software component designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other software components interact with the web service in a manner prescribed by its interface description using SOAP messages.

Web services are self-contained and loosely coupled

Web services are self-contained and loosely coupled software entities. They can be published, located and invoked across the web. Web services offer mechanisms for building interoperable, distributed, platform and language-independent systems. They lend themselves naturally to incorporation into the SOA paradigm. Their features satisfy immediately the requirements that services in a service-oriented architecture should satisfy.

Web services communicate through messages

The web services framework specifies how distributed components (services) communicate in order to use other services' functionality via the Internet. Communication is based on message exchange. A web service receives a message containing a request. It will process the request and send a response message back to the requester.

XML-based standards

The entire communication infrastructure uses XML based standards: SOAP (formerly an acronym for simple object access protocol, now considered a name), WSDL (web services description language), and UDDI (universal description, discovery and integration).

SOAP

SOAP defines a common syntax for data exchange assuring syntactic interoperability [W3C 2003]. Any web application, independent of the underlying programming language, can send a SOAP message with the service name and input parameters via the Internet and will in return obtain another SOAP message with the results of this remote call.

SOAP provides an "envelope" for requests and responses

SOAP provides an "envelope" for wrapping and sending service requests and responses. SOAP messages are represented in XML format, blowing up even simple requests and responses into many lines of XML code. Fortunately it is not the programmer who has to write this code. Software tools and IDEs (integrated development environments) normally generate XML messages from higher-level service interfaces.

SOAP request example

Figure 3-7 shows a SOAP message containing a request for product information. The service consumer wishes to check how many units of the product with ID A-1088 are available in stock and to receive details of this product (e.g. description, price, description, quantity). The code was generated by a development tool (Oracle JDeveloper 10g). The actual request is to invoke the "getProductInfo" operation exposed by the "MasterDataService" web service with a "productID" parameter value of "A-1088" (all printed in bold italics in figure 3-7).

SOAP response example

The web service returns the result in another SOAP message as shown in figure 3-8: the "name" ("racing bike"), the "description" ("low-end racing bike for upward mobile professionals"), the "price" ("230.99"), and "13" as the "quantityAvailable" (all printed in bold italics in figure 3-8). The names of these elements are defined in the web service's interface.

The XML code was actually generated by the development tool from Java source code such as:

```
public Product getProductInfo(String productID){
  ...
}
```

Figure 3-7 **Web service request as a SOAP message**

```
//SOAP Request

<?xml version = '1.0' encoding = 'UTF-8'?>
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getProductInfo xmlns:ns1="MasterDataService"
         SOAP-ENV:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
         <productID xsi:type="xsd:string">A-1088</productID>
    </ns1:getProductInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3-8 **Web service response (SOAP message)**

```
//SOAP Response

<?xml version = '1.0' encoding = 'UTF-8'?>
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getProductInfoResponse xmlns:ns1="MasterDataService"
         SOAP-ENV:encodingStyle=
               "http://schemas.xmlsoap.org/soap/encoding/">
      <return xmlns:ns2="http://Products/IMasterDataService1.xsd"
              xsi:type="ns2:Products_Product">
         <id xsi:type="xsd:string" xsi:nil="true"/>
         <name xsi:type="xsd:string">racing bike</name>
         <description xsi:type="xsd:string">
             low-end racing bike for upward mobile professionals
         </description>
         <price xsi:type="xsd:double">230.99</price>
         <quantityAvailable xsi:type="xsd:int">13
         </quantityAvailable>
      </return>
    </ns1:getProductInfoResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this code, the "getProductInfo" method returns a "Product" type object. "Product" is a class declared in the Java program, containing the fields "productID", "name", "description" etc. These are the names that were used in the generation of the SOAP code.

WSDL

An immediate question is: how does the service consumer know what to send in the request, i.e. the name of the operation ("getProductInfo" in our example) and the respective parameters that are expected and returned ("productID", "name" etc.)? This information is contained in the web service's public interface, specified in WSDL.

W3C's definition of WSDL

The W3 Consortium defines WSDL as "... an XML format for describing network services as a set of endpoints operating on messages. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint" [W3C 2001].

Clients needs to know the WSDL specification

Every web service has a WSDL description specifying how to communicate with the web service. Any service consumer (client) – or more precisely, any developer of a program invoking the web service – needs to know this specification in order to employ the web service correctly in his/her program.

The WSDL description is also processed by the tool that generates the SOAP messages. When the web service is actually invoked, i.e. at runtime, the SOAP request message is sent directly to the service provider's site. The WSDL file is not needed at that time any more.

Figure 3-9 contains some selected excerpts of the WSDL file for the above example. The WSDL description is a lot more blown up than the SOAP message. Fortunately this code is generated, so it is not the programmer who has to write down all the details in XML format.

The "MasterDataService" web service exposes three operations, "getProductInfo", "getProductDetails", and "getQuantity". The parameter to be provided in the request invoking "getProductInfo" (in the message part) is "productID". The result provided as response from the web service is an object of type "Products_Product", with the name "return". This type is declared as a complex type in the upper part of figure 3-9.

UDDI

Inhouse web services

How does the service consumer know where to send the request, i.e. who is providing the web service? There are two answers to this question. The first one is: The service consumer, or more precisely, the developer of the client program, just *knows* the service provider's address. SOAP messages are sent to web addresses or URLs (uniform resource locators). If the system is developed within the organization, a place to store the web services will be defined, e.g. a project repository.

Figure 3-9 **WSDL description for "MasterDataService" webservice**

```
//WSDL

<?xml version = '1.0' encoding = 'UTF-8'?> <!-- Generated by
  Oracle JDeveloper 10g Web Services WSDL Generator -->
<definitions
  name="MasterDataService"
    ...
  <types>
    <schema
        ...
      <complexType name="Products_Product" ...>
        <all>
          <element name="id" type="string"/>
          <element name="name" type="string"/>
          <element name="price" type="double"/>
          <element name="description" type="string"/>
          <element name="quantityAvailable" type="int"/>
        </all>
      </complexType>
    </schema>
  </types>

  <message name="getProductInfo0Request">
    <part name="productID" type="xsd:string"/>
  </message>
  <message name="getProductInfo0Response">
    <part name="return" type="ns1:Products_Product"/>
  </message>
      ...

  <binding name="getProductBinding" type="tns:getProductPortType">
    <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="getProductInfo">
      <soap:operation soapAction="" style="rpc"/>
      <input name="getProductInfo0Request">
        <soap:body use="encoded" namespace="MasterDataService"
         encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output name="getProductInfo0Response">
        <soap:body use="encoded" namespace="MasterDataService"
         encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="getProductDetails">
       ...
    </operation>
    <operation name="getQuantity">
       ...
    </operation>
  </binding>

  <service name="MasterDataService">
    <port name="getProductPort" binding="tns:getProductBinding">
      <soap:address location="http://..."/>
    </port>
  </service>
</definitions>
```

The developer will then address this location to invoke the web service. Most web services today are used in this way, since the majority of SOA based systems are still inhouse systems.

Yellow Pages

The second answer is: The developer will *look up* the provider in something like the Yellow Pages. The general idea underlying web services is to make such services available to anyone interested in the service via the Internet. For this purpose, a common point of reference, or a directory (or many directories), is required. Assuming that such a point of reference is available, developers can look up who provides the web service they need for their work.

UDDI: making web services publicly known

UDDI (universal description, discovery and integration) is one approach to making web services publicly known and accessible. UDDI defines a way to publish and discover information about web services. It is a platform-independent, open framework for describing services, discovering businesses and integrating business services using the Internet [Newcomer 2004, ch. 3]. The UDDI approach relies upon a distributed registry of organizations and their service descriptions implemented in a common XML format.

Public and private UDDI registries

UDDI registries can be public or private registries. A private registry is only accessible within a single organization or by a well-defined set of users. The public registries were originally intended as a logically centralized, physically distributed service that replicate data with each other on a regular basis. When an organization registers with a single instance of a public UDDI registry, the data is automatically shared with other public UDDI registries and becomes freely available to anyone who needs to find web services. Any organization may look up services in a public registry using a SOAP call and will obtain a list of services that meet the given criteria.

Decline of public UDDI registries

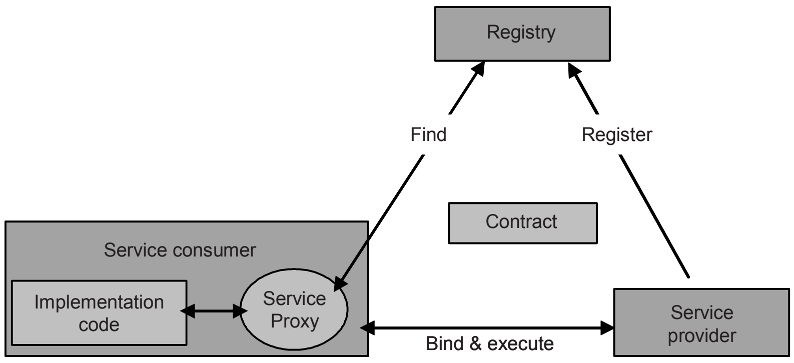
In the beginning of the SOA age, a number of public UDDI registries were set up. Eventually most of them discontinued to operate as the UDDI service was integrated into commercial products such as development tools, IDEs, platforms and servers.

3.3.2 Web Services as Building Blocks of a SOA

Find, bind and execute

Providers may register their services in a registry. The simple scheme of requests and responses as shown in figure 3-6 has to be extended when a service consumer first has to search for suitable web services on the Internet. If both the consumer and the provider agree on the terms, the consumer can then use a registered service. This triangle has been called a "find, bind and execute" paradigm. It is illustrated in figure 3-10. Six entities are involved [McGovern 2003, p. 37]: the service consumer, the service provider, the service registry, a service contract, a service proxy and the service lease.

Figure 3-10 **SOA entities and "find-bind-execute" paradigm§**



SOA entities

- The *service consumer* is a software component (e.g. another service) that requests a service. The consumer looks up the service in the registry, finds out about the terms of use and the location of the service, and initiates the process of binding to the service and executing the requested operation.
- The *service provider* is a network-addressable component that accepts and executes requests from consumers. The provider publishes a service contract in the registry that potential service consumers have to comply with.
- The *service registry* is a directory on the network that stores information about web services (such as name, description, provider, WSDL file and contracts) from service providers and displays this information to any interested party.
- A *service contract* is a specification describing the interactions between a service provider and a consumer. It may also specify pre-

§ Cf. McGovern 2003, pp. 37, 39.

and postconditions for service execution or quality of service (QoS) levels. For example, a QoS attribute may be the time it takes to execute a service method.

– The *service lease* restricts the time for which a contract is valid, i.e. the time from the beginning of the contract to the time specified by the lease.

– A *service proxy* is an additional entity that helps the consumer execute a service by calling a proxy function instead of accessing the service directly.

Most web services today are not registered

These entities describe a general framework for web service consumers, providers and registries. However, most web services used today are services that are known beforehand (e.g. developed in the same organization or by business partners). They are invoked directly by the service consumer, so a registry is not involved and contracts and leases don't need to be considered explicitly.

Web services are interoperable, supporting different platforms and languages, usually coarse-grained, and network addressable. The flexibility of a service-oriented architecture comes largely from the fact that web services are modular, composable, location-transparent, self-contained, dynamically bound and loosely coupled. Modular and composable means that services can be aggregated into composite services or into a larger solution with a limited number of known dependencies.

Web services are loosely coupled

The concept of *loose coupling* aims at the minimization of dependencies between modules of a system. Loose coupling is an important quality attribute of any software architecture. In a SOA loose coupling is accomplished through the concepts of bindings and contracts [McGovern 2003, p. 49]. When a consumer wishes to use a web service, it binds the request message to a transport type that the service accepts and sends the message over the transport to the service provider. The provider executes the requested function and returns a message whose format is specified in the service description (WSDL). The coupling is loose because the only dependency between the provider and the consumer is the binding to the service based on the interface specification in the WSDL description.

Middleware: enterprise service bus (ESB)

In a service-oriented architecture, a middleware functioning as a mediator between service consumers and service providers can be used. This middleware – an enterprise service bus (ESB) – facilitates the invocation of services. It provides additional functionality such as transforming message formats between consumers and providers, converting protocols, and routing requests to the correct service provider [Endrei

2004, p. 41]. An enterprise service bus is usually based on a messaging system.

Figure 3-11 **Service-oriented architecture with an enterprise service bus**

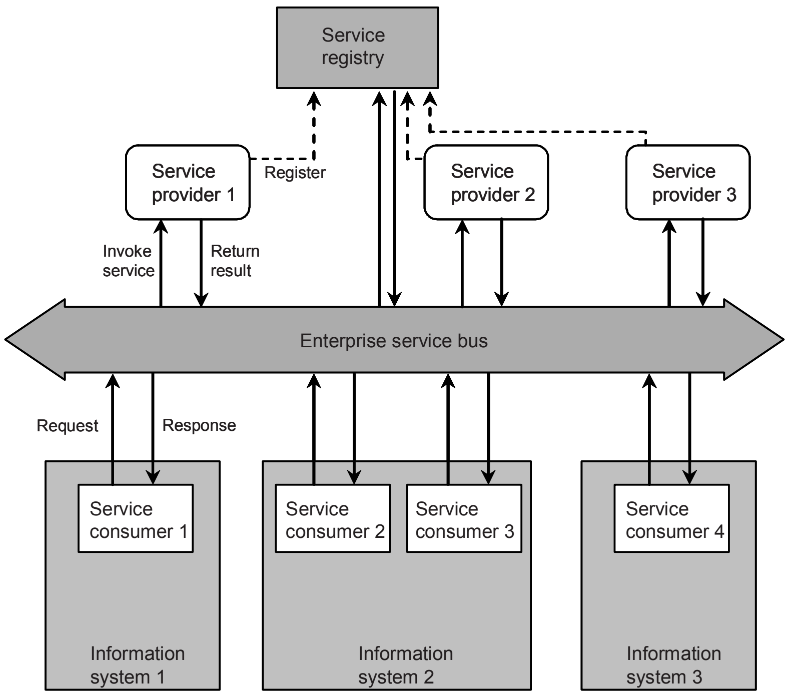


Figure 3-11 illustrates the middleware concept. Ideally the service consumers – different information systems, modules of one system, or other services – communicate only with the enterprise service bus, giving requests to the service bus and taking responses from the bus. Finding appropriate services, if such services are not known beforehand, and perhaps agreeing on terms with the provider's organization is still left to the developers creating the service consumers.

Web services tend to expose low-level functions

Web services tend to be fairly atomic, exposing relatively low-level functions – as opposed to the business functions or business-process steps the services are intended to automate. This means that either services on a higher abstraction level, which is closer to the business application, should be provided or that low-level web services have to be aggregated, i.e. combined into composite services. Composing web services, also called *web services orchestration (WSO),* is an aspect of software reuse that will be discussed in section 4.4.2 on reuse-oriented process models. Providing higher-level services is discussed in the next section.

3.4 Enterprise Service-oriented Architecture

An enterprise SOA (ESOA) is a service-oriented architecture on an abstraction level which is closer to the business problems. An ESOA makes use of enterprise services.

3.4.1 Enterprise Services

Enterprise services are services that automate business problems. The term enterprise service existed before web services became popular, in particular in the context of enterprise messaging systems and the enterprise-service-bus concept [Chappell 2004]. Nowadays, enterprise services stand for a type of web services, namely web services on the business level. This meaning of the term has been particularly stressed by SAP since they introduced the SOA approach for their new products.

Enterprise services are abstractions of business activities

While web services are often fine-grained, exposing some functionality delivered by a single information system or a module of such a system, enterprise services ideally are abstractions of business activities, not of software systems or modules. Enterprise services are defined at a granularity where they can be understood by business analysts, and therefore not require a developer to translate. Since business activities are part of business processes and processes often go across business functions, an enterprise service is likely to employ functionality from different information systems, modules or web services.

Enterprise services are web services on a business level

Enterprise services use web-service technology (XML, WSDL, SOAP etc.), therefore they can be looked at as web services on the business level. Being on a higher abstraction level means also that enterprise services are more powerful than web services, often composed of other enterprise services and/or web services. Enterprise services are defined by SAP as "an aggregation of fine-grained web services in combination with simple business logic" [SAP 2007a, p. 2].

Composite application

SAP's concept of enterprise services is that these services can be combined to form *composite applications*. A composite application composes functionality and information from existing systems to support new business processes or scenarios [SAP 2005a, p. 3]. The conceptual level of enterprise services is intended to be such that a business analyst can "assemble" enterprise services into composite applications that enable new business scenarios [SAP 2004a, p. 16].

Enterprise service example

The difference between a web service, on the single-system level, and a business-level enterprise service can be demonstrated by the following scenario [SAP 2006b, p. 7]:

Consider a business-process step such as cancelling an order that originated in the finance department in response to a customer's credit standing. Carrying out the task takes more than the single deletion of the order record in the sales management system. From a business perspective, several activities across business functions and across information systems are needed, including sending a confirmation to the customer, removing the order from the production plan, releasing materials allocated to the order, notifying the invoicing department, and changing the order status to "inactive" or deleting it from various systems.

For each of these activities, a single web service might be offered from the different systems (or from the modules of the company's ERP system) involved. If just these web services are provided, an employee responsible for the cancellation of the order will have to go to each system or module, i.e. start a screen, and carry out the necessary action.

An enterprise service would combine the tasks solved by the various web services and the employee's steps into one service. The employee would just initiate the process, e.g. start a screen that leads to invocation of the enterprise service "Cancel order".

Enterprise services can be used as building blocks for larger solutions

Complex end-to-end solutions like this can be composed with the help of enterprise services, both in the development of new and the reuse of existing information systems. Enterprise services can be reused in different contexts. Thus they are the building blocks for creating larger solutions, based on existing and on new components. They can be assembled to compose new systems and enable new business processes. Being platform and language independent, they can also be used to communicate business logic between software systems running on disparate platforms [SAP 2006b, p. 8].

3.4.2 Key Features of Enterprise SOA (ESOA)

ESOA is an enterprise-level approach to SOA

SAP calls ESOA its blueprint of a service-oriented architecture (SOA) – "a business-driven, enterprise-level approach to service-oriented architecture that offers increased adaptability, flexibility and openness" [SAP 2007a, p. 2]. At the heart of ESOA are enterprise services. ESOA extends the SOA concept just as enterprise services enhance web services, raising them to a higher level, i.e. to the business level.

Since business systems are mostly taking a process-oriented view, the enterprise service-oriented architecture goes hand in hand with business processes. Enterprise services are the building blocks in modeling, designing and implementing new business processes and changing existing processes. Taking into account that solutions for many business processes and process steps already exist, hard-wired within conventional information systems, a goal of ESOA is to decouple business processes from the underlying systems so that process steps can be added, removed or changed – even without interrupting daily operations.

Abstraction from the service's implementation

In other words, an enterprise service provides a high-level interface isolating the functionality interesting for a service consumer from the service's implementation. This abstraction is helpful to combine and recombine functionality from different applications as needed – and without having to pull existing solutions apart and start all over [SAP 2006b, p. 8].

Architecture of an ESOA based information system

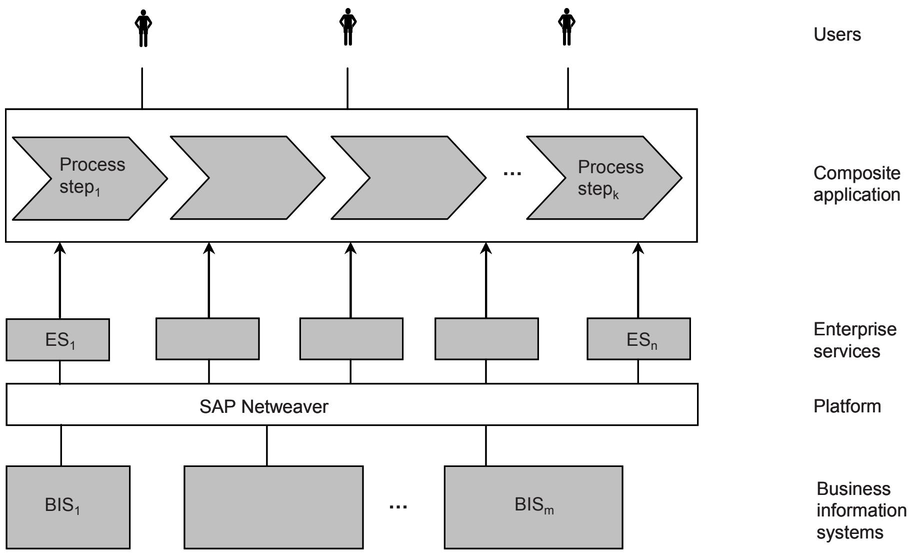
A typical architecture of an ESOA based information system has four layers as illustrated in figure 3-12. The bottom layer contains the company's existing information systems such as an ERP system and a CRM system. These systems expose reusable functionality as enterprise services with the help of SAP's NetWeaver platform (see section 3.5).

Composite applications

ES aggregated into composite applications

Enterprise services are aggregated into so-called *composite applications*. A composite application is defined by SAP as making use of data and functions provided as services by underlying systems.

Figure 3-12 **Enterprise service-oriented architecture (ESOA)**



A composite application combines functions and data into a coherent business scenario, supported by its own business logic and specific user interfaces. Such a composite application may use functionalities from many modules of the underlying information systems, e.g. SAP systems, third-party systems and inhouse legacy systems. Enterprise services manage and control the flow of information from one information system to the next, and from one department to the next.

Example: product change request (PCR)

As an example of a composite application, SAP outlines the automation of a business process regarding a product change request (PCR). The following description is adopted from [SAP 2004a, p. 11].

A PCR process is initiated when an important part of the product needs to be changed. Reasons for this can be that the design, the materials used to build the product, the needs of a particular customer, or regulatory requirements have changed. In all cases, the entire manufacturing process must be examined, and many reviewers are involved. The standard operating procedure for PCRs at most firms is paper based. Each reviewer examines the PCR document and gathers information from the underlying information systems, entering and reentering data as needed. In the end, the required approvals are all obtained.

A composite application for PCR

A composite application for PCR based on enterprise services would replace the paper request with a set of interactive forms. The reviewers enter the required information into such forms. The forms are submitted and the information is automatically transferred into the underlying information systems. Much of the information in the forms is automatically populated with data from the underlying systems. Some of it has been entered by previous reviewers further up the approval chain, avoiding additional entry and reentry of data. Enterprise services move the data back and forth from the many different systems needed to populate one form. All of the information moves intact from one approver to the next.

In an architecture like the one in figure 3-12, service consumers and service providers are typically – but not necessarily – within one organization. Finding services and terms of use (contract) are not an issue in this case, so a registry is not needed. Many business processes go across an organization's boundaries, extending to suppliers or customers. In this case, the bottom layer of the architecture includes information systems of business partners as well, just as some of the enterprise services may be provided by the business partners.

Since SAP has many installations and customers, it can be expected that SAP's enterprise service-oriented architecture will be widely disseminated. Not only is SAP migrating their own standard software onto that architecture; customers are encouraged to build their custom systems around the SAP software with the help of ESOA technology as well. Enterprise SOA is intended to become the architecture for a customer's entire information systems landscape.

Developers can create end-to-end enterprise services

The strong promotion of enterprise services includes assistance for developers with a so-called "inventory" of enterprise services. Developers can take the smaller services available in the inventory and link them together to create new systems, e.g. end-to-end enterprise services that support complete processes.

Eventually an enterprise services registry will be provided for selected partners and customers. The abstraction level is supposed to be raised eventually to a level on which business analysts are able to create enterprise services themselves – with the help of a high-level modeling tool that enables them to link services, without the need for programmers [SAP 2006b, p. 11].

Benefits of ESOA

The benefits of an enterprise service-oriented architecture extend the advantages obtained from a service-oriented architecture (SOA). With enterprise services built on top of existing information systems, these systems can be used in a flexible way and reused for newly configured solutions in the future. Because of their high abstraction level, it is easier for business analysts to understand and model enterprise services than plain web services.

Another advantage of an enterprise service-oriented architecture is that developers do not need to deal with the semantic interoperability between web services created on different systems. ESOA tools resolve the data and process disparities between different web services.

Information hiding in enterprise services

Information hiding is ensured just as in web services. Composite applications that use an enterprise service are not affected by changes in the underlying information systems. This is contrary to a conventional system architecture. When an individual piece of application functionality in such an architecture is changed, all interfaces and applications that touched the component have to be changed as well.

The most important benefits of ESOA promised by SAP are speed and flexibility through efficient aggregation and reuse of IS functionality.

Many innovations in business models and business concepts are only possible if customized IS solutions supporting the innovation are available. However, a standard software package is unlikely to provide just that piece of functionality that is needed for the specific innovation. Therefore new solutions are often developed from scratch.

Building new, individual solutions from scratch is expensive

Building new, customized solutions that support innovation is expensive and time-consuming because some of the functionality of the existing package will probably be rebuilt. Later, as the innovation becomes a standard practice, the custom-built solution has to be integrated with, or migrated into, the standard package. However, because the custom solution and the package are usually based on different platforms, the transition tends to be a costly and lengthy process. The consequence is that conventional IS solutions stimulated by an innovation cannot be delivered at an appropriate speed and cost [SAP 2005a, pp. 1-2].

New solutions based on an enterprise service-oriented architecture, on the other hand, benefit from reusable services and, perhaps more importantly, from immanent integration of these services with the existing information systems. Instead of building the custom solution from scratch, isolated from the company's other back-office systems, and with partly redundant functionality, the new solution will be based on the same technology and seamlessly integrated in the overall information systems architecture of the organization.

3.5 Platforms

Platforms provide the technological infrastructure for information systems

A platform provides the technological infrastructure for an organization's information systems. While an architecture prescribes a general pattern for the arrangement of the elements of information systems and for the interaction of these elements, a platform defines how and through which kinds of software the computer hardware is operated. A platform provides tools and mechanisms to develop programs and to execute programs. There are several levels on which the term platform is used:

Hardware platform

- A *hardware platform* is the set of hardware components that make up a specific type of computer system for which basic operating-system software is written. Examples of hardware platforms are PCs with Intel processors, Sun SPARC workstations and IBM AS/400 midrange computers. Any hardware platform requires specific systems software to make use of the hardware components.

Hardware and software platform

- A *hardware and software platform* is composed of a hardware platform and the system software (operating system, networking components, graphical user interface components etc.) written for that hardware. For users of the platform, the system software determines what application software can run on the platform and how this is done. Examples of hardware and software platforms are PCs with Windows; Sun workstations with Solaris; and Apple computers with OS X. This type of platform is losing importance, since operating systems and hardware systems are increasingly being decoupled.

Software platform

- A *software platform* is the set of basic software components that determines how other software can be developed, executed and provided to users. A software platform runs on top of a hardware platform, completely abstracting from any particular hardware. An example of a software platform is the Java platform. It runs on different hardware and software platforms.

With regard to software, the term platform is actually used on different abstractions levels. For example, an application server is called a platform because it provides the basic infrastructure for developing and deploying network-based multi-user software systems. Java EE, a set of higher-level Java components, is called a platform because it comprises all the functionality needed to create and run enterprise information systems.

A software platform defines technological features and restrictions

In the context of making information systems, software platforms play an important role. The available platform defines which technological features can be used and what the restrictions are. In particular, the platform determines the way in which an architecture can be implemented; which tools are available for the development, deployment and operation of information systems (e.g. IDEs, version control tools, application servers); and how components of the architecture can be added and removed.

Important platforms today

Although there are a variety of software platforms (and numerous products called platforms), some may be considered more important than others for the majority of today's organizations. They are fundamental in the sense that they prescribe certain ways of developing, integrating, and executing software. Important platforms today include the following:

– Java platform
– Microsoft .NET
– SAP NetWeaver
– IBM WebSphere
– LAMP (Linux; Apache, MySQL; Perl, Python, PHP)

3.5.1 Java Platform

The Java platform is a very comprehensive platform. It comes in three variants which are called editions:

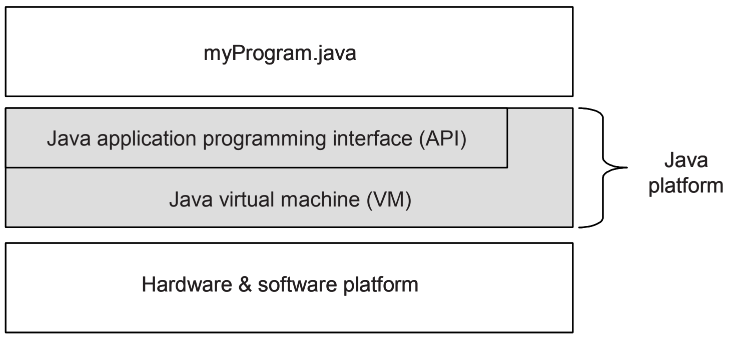
– Java ME (micro edition)
– Java SE (standard edition)
– Java EE (enterprise edition)

Former names: J2ME, J2SE and J2EE

Before 2006, the editions were known as J2ME, J2SE and J2EE (J2 = Java 2 platform). Java ME is used on small devices such as cellular phones, PDAs and pagers. Java SE comprises the essential tools for developing, deploying and running web-based and conventional information systems in Java. Java EE is an extension of Java SE supporting distributed multi-tier enterprise systems.

The Java platform determines how Java programs are developed and executed. Located between the user's Java program and the underlying hardware and software, the main parts of the platform are the Java virtual machine (Java VM) and the Java application programming interface (Java API). These parts are shown in figure 3-13 in a simplified view:

Figure 3-13 **The role of the Java platform [Campione 2001, p. 4]**


myProgram.java
Java application programming interface (API)
Java virtual machine (VM)
Java platform
Hardware & software platform

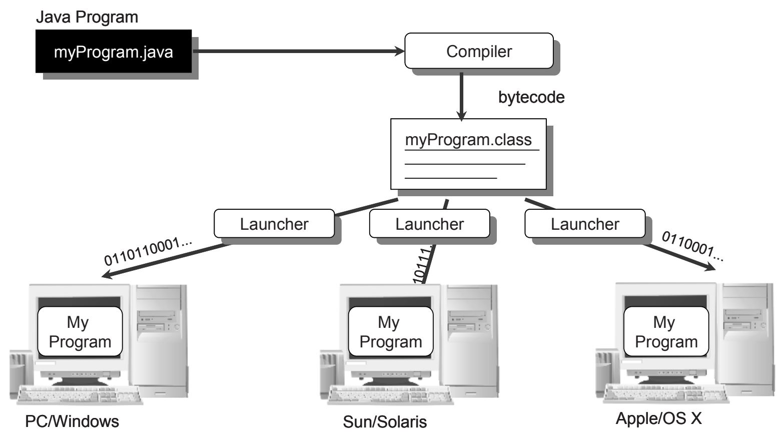
The Java VM is a "virtual" computer that can translate and execute Java programs. The Java API consists of a large collection of ready-made software components, grouped into libraries and/or packages of related classes and interfaces. Examples are packages for the development of graphical user interface (GUI), applets, concurrent programs etc.

Translating the Java programs

The Java platform is independent of hardware and operating systems, i.e. it can run on different hardware and software platforms. This independence is achieved through a two-level process in which Java programs are translated twice. Figure 3-14 illustrates these steps. A Java compiler translates the Java program into an intermediate language, so-called *Java bytecode*. This bytecode is independent of a hardware and/or software platform.

The *Java launcher* tool then translates the bytecode and runs it with an instance of the Java virtual machine. Java VMs are available for all major platforms. Each VM can take the same bytecode, translate it for the respective hardware and execute it on that hardware. Java bytecode can be interpreted or compiled. Launchers include optimization features which first compile bytecode into native code and then adaptively optimize the native code according to the runtime characteristics of the program.

Figure 3-14 **Compiling and launching Java programs**[§]



Java SE (Java platform, standard edition)

Java development kit (JDK)

The standard edition comprises a large number of programming interfaces, providing tool support for the development, deployment and execution of Java programs. In Sun's official documents for the Java platform, these tools together with the Java language constitute the Java development kit (JDK). The set of software tools forming the JDK is sometimes also called the Java platform.

Components of the Java platform

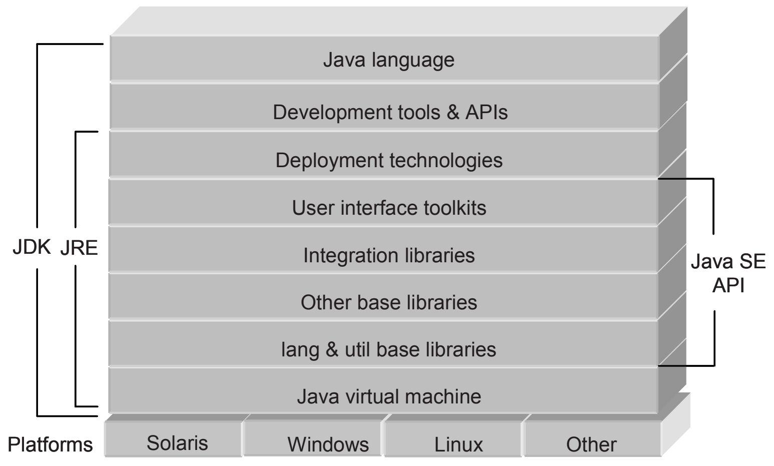
In figure 3-15, the JDK for the Java platform, standard edition (official name: "Java™ Platform, Standard Edition 6") is outlined. According to Sun's numbering scheme, the JDK is referred to as "Java™ SE

§ Adapted from Campione 2001, p. 4.

Development Kit 6" (or "JDK 1.6.0"). A subset of the JDK is the Java runtime environment (JRE). In figure 3-15, the application programming interfaces are combined into the "Java SE API". As shown in the figure, the Java platform includes:

– the Java language,
– development tools and APIs (e.g. compiler, launcher),
– deployment technologies (e.g. web-based deployment),
– user interface toolkits,
– integration and other libraries,
– language and utilities libraries,
– the Java virtual machine (VM).

Figure 3-15 **Java platform, standard edition [Sun 2006b]**



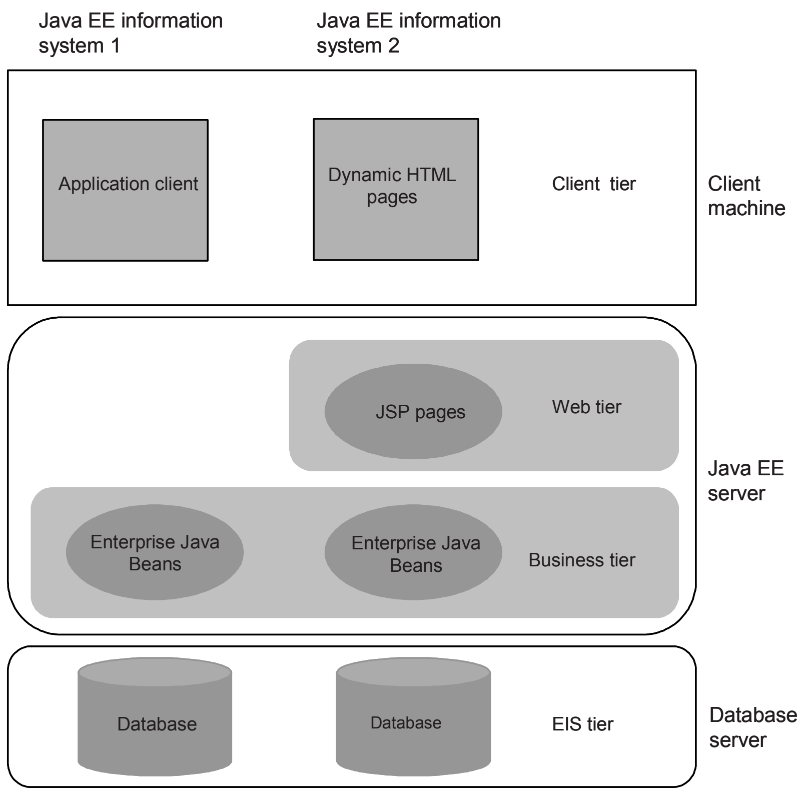
(It should be noted that the terms APIs, toolkit and technology are sometimes used interchangeably in Sun's documents.) Details of all tools comprised in the JDK can be found in the JDK documentation [Sun 2006b].

Java EE (Java platform, enterprise edition)

Most heavy-weight real-world business information systems today are distributed systems with a multi-tier architecture, allowing access and processing transactions by many users at the same time. The Java platform supports the development and deployment of such systems with components that go beyond the Java SE components.

Java EE, better known by its former name J2EE ("Java 2 platform, enterprise edition"), comprises all components of the standard edition plus additional components for heavy-weight distributed multi-tier enterprise systems. Java EE assumes that information systems have a four-tier architecture as shown in figure 3-16.

Figure 3-16 **Example of two Java EE-based IS [Jendrock 2007, ch. 1]**

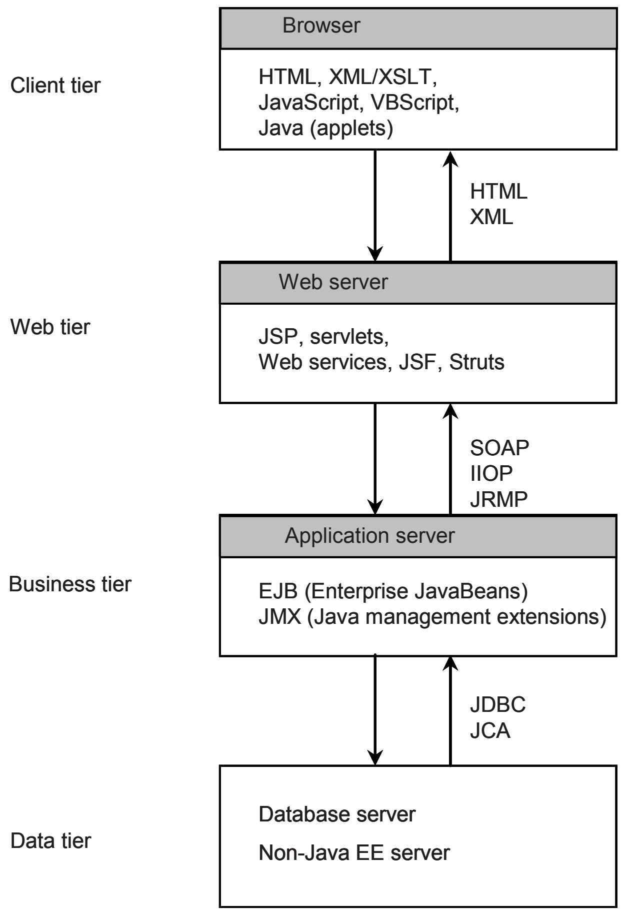


Java EE assumes a four-tier IS architecture

This logical architecture is often reduced to a physical architecture of three tiers because the web-tier and the business-tier components usually reside on one machine (Java EE server). The client-tier components run on the client computer, and the components of the so-called EIS tier run on a database server and perhaps other servers. EIS stands here for "enterprise information systems", including databases and so-called legacy systems (i.e. systems not based on Java EE technology).

In figure 3-16, information system 2 is a web-based system with a browser front-end (HTML), using JSP (JavaServer Pages) technology to generate dynamic page content. Information system 1 has a graphical user interface created with Java technology (Swing, AWT etc.). Typical technologies in a web-based system used on and between the four tiers are shown in figure 3-17.

Figure 3-17 Common technologies in a Java EE four-tier architecture


Client tier
Browser
HTML, XML/XSLT, JavaScript, VBScript, Java (applets)
HTML
XML
Web tier
Web server
JSP, servlets, Web services, JSF, Struts
SOAP
IIOP
JRMP
Business tier
Application server
EJB (Enterprise JavaBeans)
JMX (Java management extensions)
JDBC
JCA
Data tier
Database server
Non-Java EE server

- *Client tier:* A browser visualizes static and dynamic web page content described in a markup language, either created by client-side technologies such as HTML, XML/XSLT (eXtensible stylesheet language transformations), JavaScript, VBScript and Java (for applets) or prepared on the web server by server-side technologies.

Client tier

- *Web tier:* A web server accepts requests from the client and prepares responses with the help of server-side technologies such as JSP (JavaServer Pages), servlets, web services, JSF (JavaServer Faces) or Struts. JSF and Struts are frameworks for separating logic and data on the web tier. Although any Java EE-compatible web server may be used, the most common web server in Java EE-based systems is Apache Tomcat [Apache 2006c].

Web tier

- *Business tier:* An application server contains the business logic which is usually implemented with the help of EJBs (Enterprise JavaBeans). Common application servers used with Java EE are IBM WebSphere, BEA WebLogic and JBoss Application Server.

Business tier

- *Data tier (database tier, EIS tier):* A database server stores data sent from the business tier in a database and retrieves data from the database upon requests by the business tier. Data exchange with non-Java EE systems is supported.

Data tier (database tier, EIS tier)

Communication between tiers

The client tier communicates with the web tier in HTML or XML. The web tier talks to the business tier with the help of middleware such as web services, CORBA (common object request broker architecture [OMG 2007]) and RMI (remote method invocation [Sun 2004]). CORBA uses the IIOP (Internet inter-ORB protocol), RMI uses JRMP (Java remote method protocol) and IIOP, and web services use SOAP. The database tier has no specific Java EE components, but an interface is provided via JDBC (Java database connectivity). Any database management system can be connected provided that it has a JDBC driver. Non-Java EE systems (e.g. an ERP or CRM system with a different technology) can be connected with the help of JCA (Java connector architecture).

Java EE supports the development, deployment and execution of information systems with a four-tier architecture such as the one shown in figure 3-17.

Component models

For the client tier, the web tier and the business tier of such an architecture, *component models* are available. Component models provide program libraries for application development. Developers use the respective predefined classes and interfaces to create their particular application components. The Java EE specification defines the following component models:

– Client-tier components: Java application clients and applets
– Web-tier components: JavaServer Pages (JSP) and servlets
– Business-tier components: Enterprise JavaBeans (EJBs)

Java EE components have to conform to specific rules and conditions

Java EE programs are made of components. These components are written in Java and are compiled in the same way as any other program. The difference between components and "standard" Java classes is that Java EE components have to conform to specific rules and conditions. They are assembled into a Java EE-based information system, are verified to be well formed and in compliance with the Java EE specification, and are deployed to production, where they are run and managed by the Java EE server [Jendroch 2007, ch. 1].

Application client

An *application client* is a Java program running on a client machine. Typically an application client is used for tasks that require more functionality and interaction than can be provided by a markup language. It has a graphical user interface created with Java technology (Swing, AWT packages).

Applet

An *applet* is a Java program that is executed by the Java virtual machine installed within the web browser on the client computer.

Servlet

Servlets are server-side Java programs that dynamically process requests and build responses on the web server. Servlets are a means to enable dynamic content in a static markup document.

JavaServer Pages (JSP)

JavaServer Pages (JSPs) are text-based files that include markup text, Java code and JavaBean components. (JavaBeans are also components but not considered Java EE components.) JSPs are an extension of the servlet components facilitating the creation of static content. When a JSP page is requested by the client, the web server compiles it into a servlet. The browser then invokes this servlet that creates the content to be sent back to the client.

Enterprise Java beans (EJBs)

Enterprise JavaBeans (EJBs) are the most powerful components for developing business information systems. EJBs provide a distributed component model for developing secure, scalable, transactional and multi-user components. EJBs are reusable software entities containing business logic. They also isolate the business logic from lower-level tasks such as transaction management and security authorization, thus the developer can concentrate on the business problem and is relieved of system programming. From a technical point of view, EJBs are standardized and allow any component complying with the rules of the EJB specification to run on any Java EE application server.

There are several types of EJBs: session beans (stateless and stateful session beans), entity beans and message-driven beans.

Session beans

Session beans are associated with client sessions. This means that the lifespan of a session bean ends when the session is terminated. Depending on how object states are treated, beans can be stateful or stateless. A *stateful* session bean holds the state of the client across invocations. This means that data (i.e. values of instance variables) are preserved between different calls of a method. A *stateless* bean, on the other hand, does not preserve data between method calls. Once a method has been executed, the data associated with that particular method call is lost.

Entity beans

Entity beans represent persistent data objects stored in a database. They provide an object-oriented mapping of the rows of a database table to corresponding objects of a Java program. Examples of entity beans are objects such as customers, invoices, accounts, machines, employees etc. Entity beans are called by session beans. For example, a session bean "order entry" will probably call an entity bean "customer order".

Message-driven beans

Message-driven beans play an important role in today's message-oriented systems. Not only web services but also other service-oriented systems send messages, e.g. via a service bus that is based on a messaging system. Messages are asynchronous by their nature whereas method invocations are usually synchronous. A session bean that is expected to do something may not be running when a message requesting the functionality arrives, so a message-driven bean has to activate the session bean or create one. Message-driven beans are not invoked by method calls but only by sending them messages.

Example of beans interacting

An example of how the different types of beans interact is presented in figure 3-18. Session beans may be invoked by any web-tier component or client that needs business-tier functionality. A message-based request coming from a messaging client is processed by a message-driven bean which in turn invokes a session bean. Persistent data are finally stored in and retrieved from a database, therefore entity beans are called by the session beans to access these data.

Components run within containers

Components are not run by themselves but instead within so-called containers. The motivation for using containers is that thin-client multi-tier applications are in fact hard to write. They involve rather complicated code to handle transaction and state management, multi-threading, resource pooling and other complex low-level details [Jendrock 2007, ch. 1]. Containers provide prefabricated solutions to all those problems, relieving the developer from writing intricate low-level code.

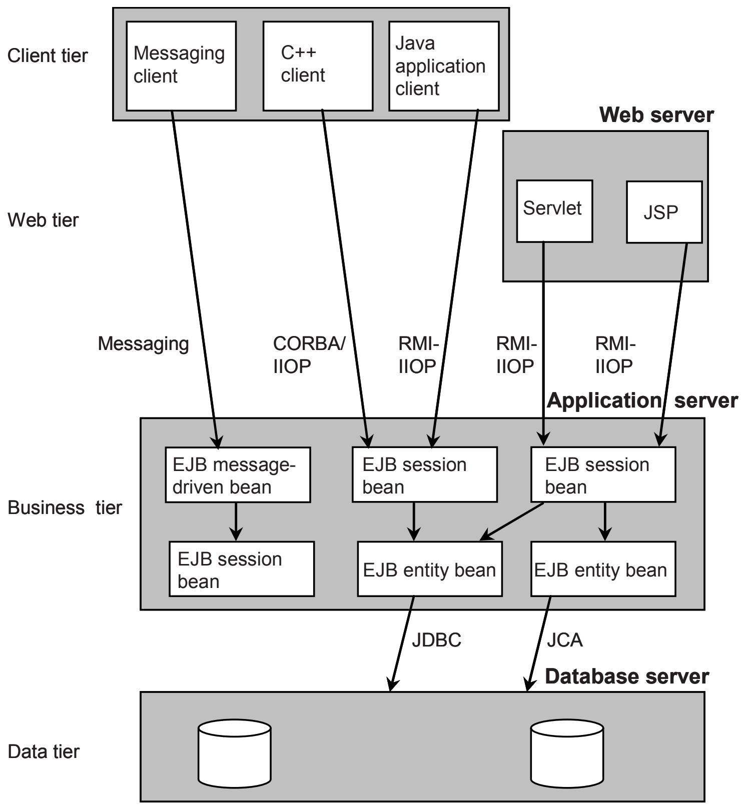
Containers provide an interface to low-level platform-specific functionality

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a Java EE component can be executed, it must be assembled into a Java EE module and deployed into its container. Containers are defined for the following components:

Container types

– An *applet container* manages the execution of applets. It consists of a web browser and a Java plug-in running together on the client.
– An *application-client container* is used for executing standard Java application clients. Both the application client and the container run on the client.

Figure 3-18 **Java EE components, servers and tiers (example)**

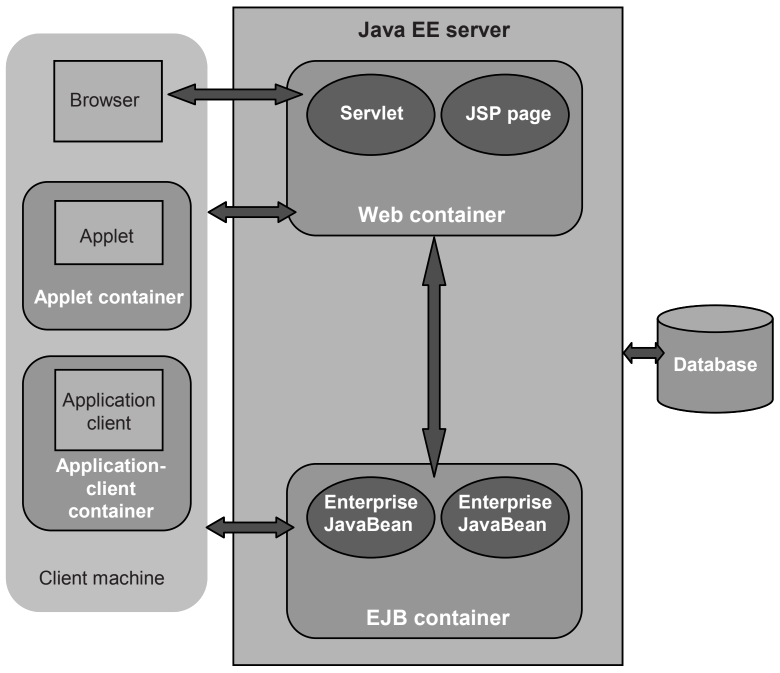


– A *web container* hosts JavaServer Pages and servlets. It manages the execution of these components. The container and the components run on the web server.

– An *EJB container* hosts the EJB components. EJBs and their container run on the application server. The main advantage of using an EJB container is that container services are available for component pooling, bean lifecycle management, client session management, database connection pooling, transaction management, persistence, authentication and access control (for details see [Singh 2002, pp. 135-136]).

Java EE components and their containers are summarized in figure 3-19. The browser on the client is responsible for both executing applets and displaying pages created by servlets and JSPs. EJBs are called by servlets, JSPs and application clients.

Figure 3-19 **Java EE containers and components [Jendrock 2007, ch. 1]**



Java IDEs

Integrated development environments (IDEs) are available for Java just as for other common programming languages or platforms. They facilitate the development of Java programs through powerful tools. However, since the Java platform is very comprehensive and Java is a rather heavy-weight language, IDEs are also heavy-weight products. Ease of use is not quite the same as for simpler programming languages.

IDEs for Java

Both commercial and open-source IDEs are available. Some open-source IDEs were originally developed and sold by companies like IBM and Sun Microsystems and later made available freely. Well-known IDEs for Java include the following:

– NetBeans (originally developed by Sun Microsystems, in 2000 made open-source; http://www.netbeans.org)
– Sun Java Studio (also by Sun Microsystems, formerly known as "Sun Forte")
– Visual J# (as part of Visual Studio .NET; see section 3.5.2)
– JBuilder (by Borland; http://www.borland.com)
– Eclipse (originally developed by IBM, in 2001 made open-source; http://www.eclipse.org)
– WebSphere Studio Application Developer (part of IBM WebSphere Studio)

Many professional Java developers use JBuilder, Eclipse or NetBeans. While JBuilder is a commercial product that has evolved through many versions over more than a decade, Eclipse and NetBeans are popular open-source IDEs. Eclipse is nowadays governed by a consortium of members firms, among them IBM, Oracle, SAP and Borland.

Specifics of IS development in Java EE

Java EE enforces a certain architecture and software technology

The above brief outline of the comprehensive Java platform shows that this platform is not only very powerful but that it also sets clear standards and restrictions regarding further application development. A four-tier architecture as laid out in the figures 3-16 and 3-17 is presupposed, the tools and APIs are oriented towards such an architecture, and the component models guide system designers and implementers in a certain manner, restricting significantly flexibility and allowing only EJB specification-compliant constructs. The EJB container controls all invocations of EJB components, interposing itself between each method call. In other words, the container puts itself between the caller and the EJB method called. As a conclusion, information system projects based on Java EE have only limited choices regarding software architecture and technology.

Projects using the Java platform are in some ways different from conventional development projects.

No choice of programming language

First and obviously, there is no decision to be made about what might be the best programming language for the problem – there is only one on that platform. In addition, building up knowledge and expertise in using the Java platform takes significant efforts. Once proficient Java architects, designers and programmers are available, the organization is likely to put its future development projects on the same platform.

Perhaps a choice regarding the appropriate IDE is left, yet the impact of this decision is less severe than the choice of the programming language. Having experience with one IDE may mean that the same IDE is also used in the future and therefore other, better IDE options are not explored.

On the other hand, IDEs are products that compete on the market with new versions to attract developers. If a new feature useful for the current project is available in the next version of some other IDE but not contained in the IDE the organizations has used up to now, switching to that new IDE may take place.

Reinventing the wheel in Java?

Second, developing in Java implies to a significant extent the use of prefabricated solution modules (classes, interfaces, patterns etc.) and extending or adapting these modules. Frequently modules created by the developers are not built from scratch, because a lot of functionality is already available (derived from superclasses). However, it is difficult to know all the things that are already available and where to find them, i.e. which libraries, packages, classes, APIs etc. do already exist. The risk that the wheel is reinvented is therefore not negligible.

The Java platform imposes a strict corset on designers and implementers

Third, the degree of freedom left to the developer is substantially less than in other languages. The Java platform imposes a strict corset on designers and implementers. Java EE prescribes a specific architecture and its component models force the developers to proceed in a certain way. Some pressure restricting the freedom is caused by the large offering of prefabricated solutions. When such a solution is available, it is attractive to use it instead of developing a new one that will cost time and money.

Fourth, some parts of Java programs are very compact, incorporating an abundance of functionality in a few lines of code. While Java experts are used to this compactness, newcomers find it difficult to understand.

"Copy and paste" programming

Finally, the emerging trend of Java becoming "the" platform for heavy-weight information systems in all areas has a remarkable side effect: Solutions to many types of problems developed worldwide are published and can be found on the Internet. Java programmers often copy code from the Internet and paste it into their own programs. Project managers need to observe this type of "copy and paste" programming carefully because it can create problems regarding the software quality and copyright issues.

3.5.2 Microsoft .NET Platform

When .NET was announced in 2000, Microsoft called it an "initiative". For some time it remained unclear what .NET really was – a product, an architecture, a framework, or a platform? In fact, .NET became a brand for an integrated set of different Microsoft technologies, some of them entirely new, some updates of former technologies or products. From a technical point of view, .NET can be considered a platform because it provides an interlocking set of tools and mechanisms to develop programs and to execute programs. (In a quite similar way, IBM introduced the brand name WebSphere as a collection of products with a platform at its core; cf. section 3.5.3.)

Objectives of the .NET platform

When .NET was advertised and introduced into the market, the goal was to provide a common platform for developing and running enterprise-wide and Internet-based information systems. The specific *objectives* of .NET were to support:

- Distributed computing and to simplify the development of client-server and other distributed systems, based on open Internet standards (HTTP, XML, SOAP etc.).
- Componentization, i.e. building systems from software components (reuse) in a simpler way than before.
- Internet interoperability, in particular through the use of software components that reside on servers somewhere on the Internet (web services).
- Language independence; this means that components can be written in different programming languages, are easily integrated, and work smoothly together.

- Language integration on the programming level; for example, a class written in one language can inherit from a class written in a different language.
- Reliability, i.e. programs under .NET are supposed to contain fewer errors than conventional programs.
- Security by providing a security infrastructure. Security is an increasingly important problem in today's web-based environments allowing access to computers and information systems from outside via the Internet.

The .NET platform has the following parts:

1. The .NET framework
2. Developer tools
3. A set of servers
4. Client software

The core of .NET, the .NET framework, will be discussed in a subsection below. The premium tool for developers is Visual Studio .NET, a very powerful and convenient-to-use IDE for developing information systems under .NET. Servers include MS Windows server, a database server (SQL server), and a server for web-based information systems (BizTalk server). Clients run under Windows XP or Windows Vista, for example.

.NET framework

Objectives of the .NET framework

The .NET framework is the core of the platform supporting information systems development in the highly distributed environment of the Internet. According to Microsoft's documentation, its objectives are [Microsoft 2007c]:

- To provide a consistent object-oriented programming environment regardless of whether the object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts; guarantees safe execution of code, including code created by an unknown or semi-trusted third party; and eliminates the performance problems of scripted or interpreted environments.

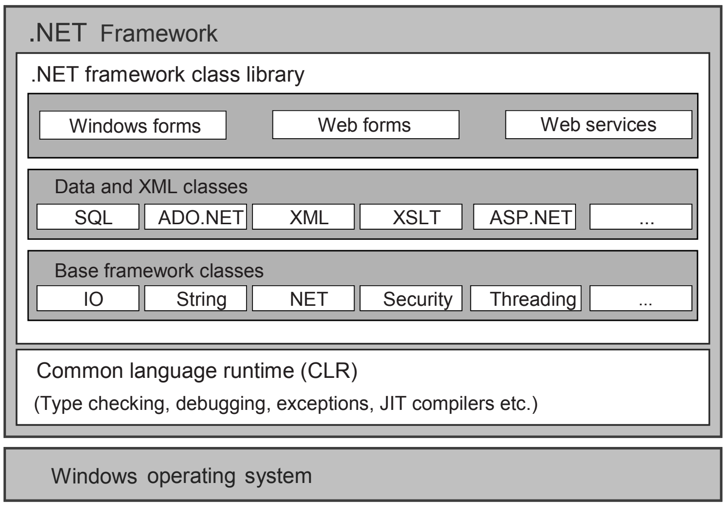
- To let developers work in the same way across widely varying types of application systems, such as Windows-based and web-based systems.
- To build all communication on industry standards to ensure that code based on the .NET framework can integrate with any other code.

The .NET framework is composed of two major parts as illustrated in figure 3-20, the common language runtime and set of class libraries.

The CLR is the foundation of the .NET framework

The common language runtime (CLR) is the foundation of the .NET framework. It manages code at execution time, providing core services such as memory management, thread management, code safety verification, compilation and other system services. The CLR is the basis for language independence and language integration. These are achieved through an intermediate-language concept.

Figure 3-20 **Components of the .NET framework**



.NET supports many programming languages

The .NET platform supports many programming languages. Next to Microsoft's core languages, Visual Basic .NET and C#, only a few more (J#, so-called "Managed C++", and JScript .NET) were available at the beginning. Today the list is long. In the MSDN (Microsoft developers network) documentation, 24 languages are listed, including APL, Cobol, Eiffel, Fortran, Java, Pascal, Perl and Smalltalk [Microsoft 2007b].

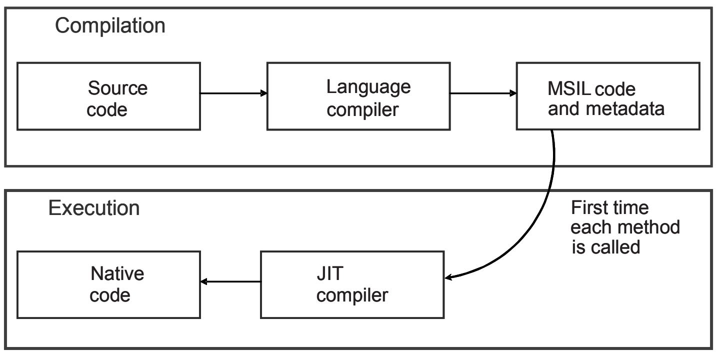
Syntactical differences between programming languages are not important

Most modern programming languages are based on similar principles and provide similar constructs. There is a common understanding in software engineering of what makes a "good" programming language (e.g. control structures from "structured programming", data types, objects, classes, inheritance etc.). But in actual fact, all programming languages look completely different due to their syntax. Some use end statements to close blocks (end if, end sub etc.), some use curly braces; some terminate statements with a certain character, some by the end of the line; etc. These syntactical differences are not really important, yet the semantics of the constructs certainly are. The idea behind an intermediate language – between the source language and the machine language – is to provide a common implementation of the semantics underlying all source languages.

MSIL (Microsoft intermediate language), JIT ("just in time") compiler

Code that is in conformance with the CLR specifications is called managed code, while code that does not target the runtime is known as unmanaged code. All managed code is compiled into an intermediate language, MSIL (Microsoft intermediate language). Compilers for .NET languages translate source code into MSIL. Before the MSIL code can actually be executed, it has to be translated into machine-specific binary code ("native code"). As shown in figure 3-21, this is done by a "just in time" (JIT) compiler. The name JIT compiler comes from the fact that the translation is done when the respective method is called for the first time.

Figure 3-21 **Translating managed code**



The CLR/intermediate-language approach is similar to the Java VM/Java bytecode as discussed in section 3.5.1. Although the Java world and .NET world are rather separate worlds (and will probably remain so), both follow the same fundamental principles and concepts. Just as Java programs can run on any hardware and software platform that has a Java virtual machine, programs in any of the .NET languages can run on any platform that has the .NET framework. However, up until now, these have been almost exclusively Windows platforms.

.NET framework class library

The .NET framework class library, the second main component of the .NET framework, is a comprehensive, object-oriented collection of reusable classes that are tightly connected with the CLR. The base framework classes provide types based on .NET's common type system (CTS). These types are used by all .NET languages that produce managed code. In addition to standard types such as integers, floating-point types, arrays etc., types supporting string management, data collection, database connectivity, file access etc. are available in the data and XML classes. The class library further includes types that can be used for the development of graphical user interfaces (Windows forms), web pages created with ASP .NET (Web forms) and web services.

Visual Studio .NET

Visual Studio .NET is a comprehensive IDE for creating any kind of software component that can run on the .NET platform, e.g. GUI forms, web forms, code modules, classes, data access components, XML files, stylesheets and more. It contains a complete set of development tools for building web-based systems with ASP .NET as well as conventional desktop-oriented systems, web services and mobile solutions.

Most code is generated

For most types of components, code is partly or completely generated from the developer's input. For example, GUI components can be entirely created by "visual programming", dragging and dropping graphical icons onto the GUI design pane. Properties of components can be set or changed in a table (properties window), access methods for user-defined classes are generated automatically, and database access (connectors, adapters, queries etc.) is provided via prefabricated classes.

For web services, SOAP messages and WSDL files are created automatically from the respective language classes. Explorers and browsers help to keep track of the logical structure ("class view", "object browser") and the physical structure of development projects ("solution explorer") including the servers involved ("server explorer").

Programming languages

The core languages included with Visual Studio are Visual Basic .NET, Visual C++ .NET, Visual C# .NET, and Visual J# .NET.

All languages share the same set of tools. In this way, access to key .NET technologies and the creation of mixed-language solutions are facilitated.

Compared to IDEs for other languages and platforms, Visual Studio .NET is not only a very powerful development environment but also one of the most convenient ones available on the market. Microsoft considers it as its flagship in the .NET world. Software developers working with Visual Studio can benefit from very powerful tools.

.NET servers

A number of Microsoft servers are available on the .NET platform, supporting important areas such as database management, electronic commerce and business process management. The .NET servers include the following products [Microsoft 2007a]:

BizTalk server

- *BizTalk server* facilitates the exchange of information among diverse information systems running on different hardware and software platforms. It is a business process management (BPM) server that enables companies to automate business processes. BizTalk server contains tools to design, develop, deploy and manage processes and to integrate processes across disparate information systems, both within the organization (enterprise application integration, EAI) and between organizations (business-to-business, B2B). BizTalk server includes mechanisms for connecting to legacy systems and to typical business packages for ERP and CRM (e.g. SAP, Siebel, PeopleSoft, Oracle and JD Edwards). A messaging engine provides a way to define and exchange XML-based documents among systems.

Commerce server

- *Commerce server:* E-commerce websites have many things in common. Instead of building everything from scratch, organizations can use the commerce server's packaged components to deploy personalized portals. Commerce servers provide features such as order processing, merchandising and catalog management with integrated search capabilities.

SQL server

- *SQL server* is Microsoft's RDBMS for distributed information systems. Clients can send queries to a database server, and the server returns the results over the network. SQL server provides enterprise data management with integrated tools for business intelligence (BI), analysis, reporting and notification.

Exchange server

- *Exchange server,* the Microsoft messaging and collaboration server, enables users to send and receive electronic mail and other forms of interactive communication through computer networks. Exchange server interoperates with Microsoft Outlook, Outlook Express and other e-mail client systems.

Content management server

- *Content management server* is an enterprise web content management system that enables companies to build, deploy and maintain highly dynamic Internet, intranet and extranet Web sites.

Other servers include the host integration server (for interoperation of Windows-based systems with IBM hosts), application center server (supporting scalability, managing replicated server applications), ISA server (Internet security and acceleration server, providing firewall and proxy services), and the speech server (for deploying and managing distributed speech applications).

Specifics of IS development on the .NET platform

Development under .NET takes place within Visual Studio

For developers, the .NET platform unfolds its power best when Visual Studio .NET is available as IDE. Visual Studio suggests an event-driven programming style, making the creation of graphical user interfaces very easy. For this reason, Visual Studio is also a powerful tool for requirements prototyping (see section 4.4.2).

Working with Visual Studio is very comfortable compared to other IDEs. The developer does not need to leave Visual Studio because everything needed in the development process is there. No matter whether a desktop-oriented information system, a web front-end, a web service, an XML file or an XML schema have to be created, Visual Studio assists the developer through tools generating code and checking whatever new code is written.

Implementation is largely controlled by Visual Studio. Developers continue to add items to the system under development, Visual Studio generates some of the code, and the developer completes the code manually. Testing and debugging are also done with the help of tools embedded in the IDE.

3.5.3 IBM WebSphere

Java-based e-business systems

WebSphere is a widely used platform for developing, deploying and running Java-based electronic-business systems. It comprises application servers, tools for Java development, connectivity mechanisms, and many more products. In fact, WebSphere is a brand name for a long series of IBM products that are related to web-based information systems, yet many people associate with the name WebSphere the best-known of these products, the WebSphere application server (WAS).

WebSphere runtime tools

WebSphere products can be divided into runtime tools and development tools. Runtime tools, next to the WebSphere application server (see further below), include the following [IBM 2008]:

WebSphere message queuing

– *WebSphere message queuing (MQ,* formerly known as IBM *MQSeries)* – enables programs to communicate with one another across a network in a messaging and queuing style: Messages sent by programs to be processed by other programs are placed in storage queues, allowing the programs to run independently of each other, at different speeds and times, in different locations, and without being connected [IBM 2003, p. 3].

WebSphere enterprise service bus (ESB)

– *WebSphere enterprise service bus (ESB)* – an abstraction layer on top of messaging, providing a connectivity infrastructure for integrating systems and services. ESB routes messages between services, converts transport protocols and message formats between requester and service, and handles business events from disparate sources.

WebSphere commerce

– *WebSphere commerce* – an e-commerce platform that supports doing business directly with consumers (business-to-consumer) or with businesses (business-to-business), and indirectly through channel partners. At the core is an online selling environment that enables companies to offer personalized, cross-channel shopping.

WebSphere portal

– *WebSphere portal* – provides a single access point to web content and IS, personalized to each user's needs; supports workflows, content management, security mechanisms and scalability.

Developments tools in and around WebSphere can be distinguished into two lines. One is WebSphere Studio (see further below), the other one is a suite from former Rational Software (now IBM): Rational application developer (RAD), Rational web developer (RWD), and Rational software architect (RSA).

WebSphere application server (WAS)

WAS connects presentation and business tiers

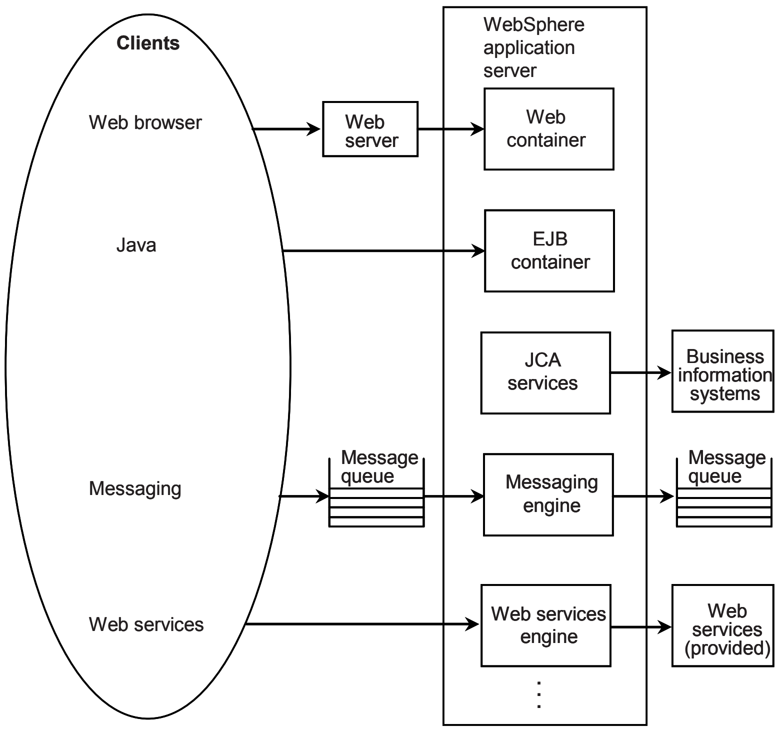
The WebSphere application server is typical middleware that connects the presentation tier and the business tier in e-business systems where clients and servers are distributed on the Internet. Clients send requests over the web. Responses are established based on business information systems (like ERP or CRM) or database management systems hosted by the WAS.

Since the web is the network on which responses and requests are transported, a web server is employed to work with the application server. In many cases this is the Apache web server, but the WAS works with other web servers, too.

WAS containers and engines

In addition to HTTP requests, EJBs, web services and messages can also invoke the application server as shown in figure 3-22. Correspondingly, the WAS architecture provides containers and engines to accept and process the respective tasks, including a web container, an EJB container, JCA services, a messaging engine and a web services engine [Sadtler 2005, p. 21]:

Figure 3-22 **WebSphere application server [Sadtler 2005, pp. 7, 21]**



– The *web container* processes HTTP requests, servlets and JSPs (JavaServer Pages).
– The *EJB container* provides all runtime services that are needed to deploy and manage Enterprise JavaBeans.
– The *JCA (Java EE connector architecture) services* provide connection management for access to business information systems. JCA specifies how connections are administered and how transactions have to be performed.
– The *messaging engine* handles and stores messages. It provides a connection point where clients can produce messages and from which clients can receive messages.
– With the help of the *web services engine,* the WebSphere application server can act as both a web service provider and as a requester. In the first case, it hosts web services that may be invoked by clients. In the latter case, it sends requests from its own information systems that need services from other locations.

The overview scheme in figure 3-22 illustrates the basic functionality of a single application server. This corresponds to a stand-alone server that hosts one or more information systems. Even though several application servers may run on one machine, they will still be stand-alone servers.

Distributed servers

In heavy-duty environments, centrally managed *distributed servers* have advantages over stand-alone servers, including workload management, scalability, failover capabilities and thus high availability. The WebSphere application server also supports distributed server configurations.

WebSphere Studio

WebSphere Studio comprises a family of IDE products for development, testing, debugging and deploying web-based information systems [Takagiwa 2002, p. 4]. It provides support for each stage of the development life cycle. WebSphere Studio is the follow-on toolset for IBM's former Java IDE, Visual Age for Java.

WebSphere Studio is based on Eclipse

The WebSphere Studio products are based on the *Eclipse workbench.* This is an open-source toolset originally designed by IBM, later released as open-source, and nowadays managed by the *Eclipse.org* consortium (http://www.eclipse.org). The Eclipse workbench provides frameworks, services and tools for building tools. Any independent software vendor can use the same APIs as IBM to create their own tools that can be plugged into the Eclipse workbench.

WebSphere Studio application developer

The products of the WebSphere Studio family provide support for a wide range of development tasks, from medium-complex websites to heavy-weight Java EE information systems based on the MVC (model-view-controller) pattern. In the middle of the range is the *WebSphere Studio application developer* – the toolset that most people associate with the name WebSphere Studio. It includes the following basic tools [Takagiwa 2002, pp. 13-18]:

- Web development tools – to create HTML pages, JSPs servlets and other resources
- Relational database tools – to create and manipulate the data design of a project in terms of relational database schemas
- XML tools – to build DTDs (data type definitions), XML schemas and XML files
- Java development tools – an IDE (integrated development environment) for Java
- Web services development tools – to build and deploy web services-enabled systems across software and hardware platforms, based on UDDI, SOAP and WSDL
- Team collaboration tools – to allow individual developers to work on a team project, share their work with others as changes are made, and access the work of other developers as the project evolves
- Integrated debugger – to detect and diagnose errors in programs running locally or remotely
- Server tools for testing and deployment – to test JSPs, servlets, HTML files and EJBs
- EJB development tools – to develop and deploy enterprise Java Beans
- Performance profiling tools – to test the performance of a system under development
- Plug-in development tools – to develop plug-ins for the Eclipse workbench

Many Java developers around the world use IBM WebSphere tools and/or Eclipse as an IDE. One remarkable feature of WebSphere is that it has interfaces with SAP NetWeaver's application platform (see immediately below).

3.5.4 SAP NetWeaver

A proprietary, high-level platform

As an example of a business-oriented, proprietary, high-level platform, *SAP NetWeaver* is presented in this section. NetWeaver is an integration and application platform both for building new, custom information systems based on SAP's technological infrastructure and for integrating an organization's existing information systems using different infrastructures. For many SAP users such a platform makes sense not just because their core information systems are based on that platform. Since the core systems dominate the organization's IT landscape, any integration or development project will have to meet the restrictions and requirements of these heavy-weight systems. This is certainly easier if all systems are using the same platform.

Motivation for SAP NetWeaver

Business challenges and requirements that led to the development of the NetWeaver platform, according to SAP, include the following [SAP 2003]: Enterprises expect their IT departments or IT organizations to make their contribution towards competitiveness, cost reduction and increasing shareholder value. Heterogeneous IT environments prevent such contributions or make them at least very difficult.

On the other hand, it is a fact that many organizations have disparate information systems and they wish to continue operating these systems in the future. No single vendor can deliver all the solutions that an enterprise needs, including SAP. A growing trend is, for example, that SAP's customers run SAP systems for their specific business processes and also use IBM and Microsoft technologies for their e-commerce solutions and office work. This means that SAP systems must be able to import, export and interoperate effectively with systems based on the .NET, Java, and WebSphere platforms. Business partners of the firm also have information systems, and since more and more business transactions are done online, integrating with those systems has be taken into account.

Interoperability between heterogeneous IT environments

Interoperability between heterogeneous IT environments has become a major issue because the cost of IT is largely determined by how well disparate information systems can be integrated. Nowadays, end-users expect seamless integration of different systems and transparent access to information from these systems, no matter where the data is actually stored. According to SAP, the lion's share of the integration effort in large businesses occurs between SAP systems and other custom business systems. Consequently integration became the major challenge to be tackled when the NetWeaver platform was designed.

SAP NetWeaver is an integration and application platform

In a simplified view, SAP NetWeaver is described as an "integration and application platform to unify and align people, information, and business processes across technologies and organizations." [SAP 2003, p. 6] In fact, NetWeaver integrates a number of technologies for different purposes: mobile, portal, collaboration, knowledge-management, business-intelligence, master data-management, business process-management, integration-broker and application-server technologies [SAP 2004b, p. 5].

NetWeaver is used to integrate information from different sources via open standards such as XML, SOAP, UDDI, WSRP (Web services for remote portlets) and WSBPEL (Web services business execution language). It is the basis for SAP's enterprise service-oriented architecture (ESOA) and the composite applications which are discussed in section 3.4. SAP's core products in the SAP business suite (including SAP ERP) are based on the NetWeaver platform as well as all new modules available from SAP partners around the world.

SAP: "integrating people, information, and business processes"

An overview of SAP NetWeaver as given by SAP is shown in figure 3-23. This overview focuses on the integration goal ("integrating people, information and business processes" [SAP 2003, p. 6]), exhibiting most major components of NetWeaver. The following description is based on SAP documents [SAP 2006a, SAP 2004b, SAP 2003].

People integration

People integration stands for bringing together the right information (i.e. the appropriate system functionality) and the right persons. NetWeaver includes four components for this purpose:

Enterprise portal

- *Enterprise portal* – provides a complete platform infrastructure along with knowledge management and collaboration software. Under a unified user interface, workers get personalized, role-based access to heterogeneous IT environments. Information can be extracted from SAP and non-SAP systems, data warehouses, web pages etc. and received from web services.

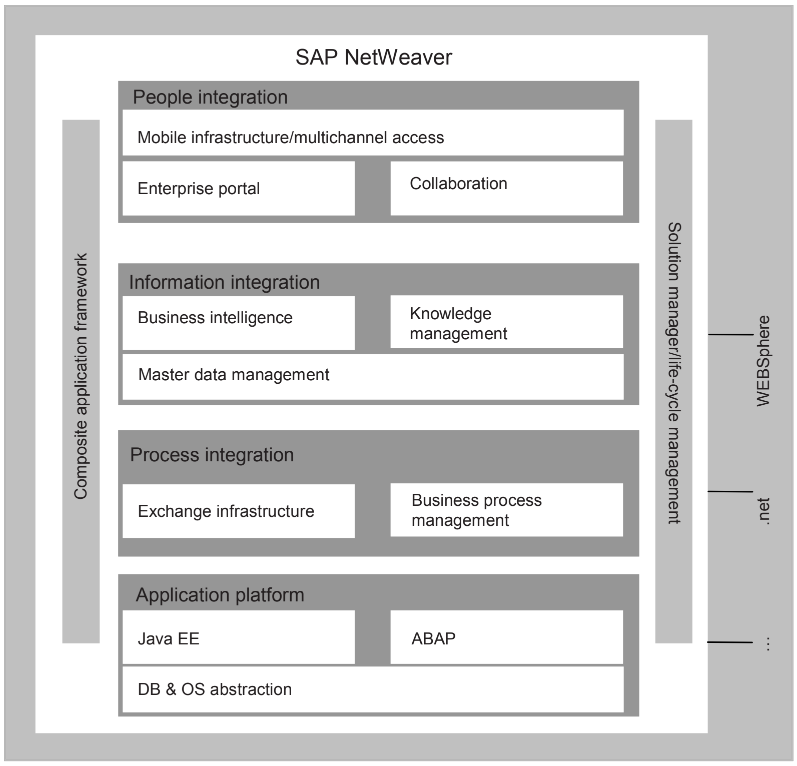
Collaboration

- *Collaboration* – supports communication among teams and communities. This includes real-time and virtual collaboration tools such as news forum, instant messaging, collaboration room, chat, team calendars, shared documents and tools etc.

Mobile infrastructure/multichannel access

- *Mobile infrastructure/multichannel access* – permits access to enterprise systems using mobile devices and voice systems, so people can stay connected any place where their business is conducted. Connections can be based on HTTP, WAP (wireless application protocol), WLAN (wireless LAN), Bluetooth, GSM (global system for mobile communications), UMTS (universal mobile telecommunications system), Voice over IP and other technologies.

Figure 3-23 **Overview of SAP NetWeaver [SAP 2003, p. 6]**



Information integration

In this category, both structured and unstructured information are made available in a consistent and accessible manner. Functionality is provided in the following areas:

Knowledge management

- *Knowledge management* – manages and makes accessible text and audio files, slide shows etc. Search features, content management, information classification and distribution, integration of external content etc. are included.

Business intelligence

- *Business intelligence* – helps to identify, integrate and analyze disparate business data from heterogeneous sources. Tools support enterprise modeling, data warehousing and data mining, queries, simulation, decision making and creating interactive reports.

Master data management

- *Master data management* – promotes information integrity across the business. Consolidation, harmonization, and central master-data management are provided, including business partner information.

Process integration

The goal of process integration is to enable efficient business processes across heterogeneous IT environments both within the boundaries of an enterprise and beyond. NetWeaver supports:

Exchange infrastructure

- *Exchange infrastructure* – providing integration technologies that support process-centric collaboration among SAP and non-SAP system components. Messages and service requests are handled based on open standards such as XML, SOAP and WSDL. Special adapters for business-to-business integration are available (i.e. processes integrating business partners).

Business process management

- *Business process management* – permitting existing information systems as well systems under planning to be combined into end-to-end business processes spanning the entire value chain. Modeling, execution and controlling of processes and workflows are supported.

Application platform

Development and runtime environment for Java EE and ABAP based software

The application platform provides a development and runtime environment for both Java EE and ABAP based software, including abstractions from the underlying operating and database management systems and a web application server as a development and deployment platform for web-based systems and web services.

More components and tools are available with the NetWeaver platform. As the time goes on, SAP provides new tools and components and realigns existing ones with new names or into new arrangements.

Composite application framework

The *composite application framework* is a development environment for building composite applications as discussed in section 3.4.2. The framework contains design tools, methodologies, services, processes, an abstraction layer for objects and user interface patterns.

Solution manager

The *solution manager* (formerly life-cycle management) provides comprehensive tools for all stages of the software life cycle: design, development, deployment, implementation, versioning, testing and ongoing operations such as administration and change management.

Auto-ID infrastructure

The *Auto-ID infrastructure* provides middleware which connects automated communication and sensing devices such as RFID readers and printers, Bluetooth devices, embedded systems and bar-code devices. RFID data can be captured, stored and transmitted so that they can be interpreted by information systems.

Interoperability with IBM and Microsoft systems

As mentioned in the beginning of this section, integration with IBM and Microsoft systems is considered a crucial requirement. Following this need, the importance of interoperability of SAP systems on the one hand and IBM and Microsoft systems on the other hand is not only stressed on a business level but also supported on a technology infrastructure level. This means that NetWeaver components for people, information and process integration as well as the application platform have counterparts in the IBM WebSphere and Microsoft .NET platforms.

Connecting with Websphere components

For example, the Java classes offering access to SAP interfaces are integrated into IBM's WebSphere Studio application developer (WSAD), and modules developed in WSAD are compatible with the runtime environment of SAP's web application server. This means that components developed in WebSphere can run under the SAP web application server.

Connecting with .NET components

Another example is the SAP .NET connector. Using this connector, SAP systems can be extended with components developed for the Microsoft .NET platform. An SAP system can access and integrate .NET services, and at the same time .NET based information systems can access SAP modules.

On the integration levels, correspondences between NetWeaver components on the one hand and WebSphere and .NET components on the other hand are established. For example, IBM's Lotus suite with powerful collaboration and information management features can be accessed from SAP systems and vice versa. A business information system running on SAP Netweaver can be integrated with an information system running on IBM WebSphere (applying JMS using MQSeries). Likewise, a Microsoft .NET solution can be connected via a MSMQ (Microsoft message queuing) adapter. In this way, user companies can manage an IT landscape in which SAP, IBM and Microsoft systems coexist.

Integration with non-SAP solutions

This is an attractive perspective for information systems development by user organizations. In many organizations, more knowledge and experience are available in general development technologies such as Java IDEs or Visual Studio .NET than in SAP specific technologies. Through the interoperability with IBM WebSphere and Microsoft .NET, organizations can easily extend and enhance their heavy-weight SAP information systems by light-weight components developed with the help of common software technologies.

3.5.5 LAMP

LAMP is a bundle of tools that are often used together

LAMP is a popular platform composed of open-source software. In fact, LAMP is not a unified platform like the above ones but a bundle of tools that are often used together. The name is an acronym that stands for:

L = Linux
A = Apache
M = MySQL
P = Perl/Python/PHP

It is attributed to Michael Kunze, who recommended this combination of products as an alternative to commercial and proprietary platforms in a German computer magazine in 1998 [Kunze 1998].

Linux

L = Linux is an open-source operating system similar to Unix. It was originally developed by a Finnish student, Linus Torvalds, in the early 1990s and given to the open-source community. Professional software firms have adopted and extended Linux, offering services and support for user organizations. Those who wish to handle Linux themselves can still download and run it for free.

Apache

A = Apache has become a generic name for a variety of open-source software products, yet the original product was the Apache HTTPD web server [Apache 2006b]. One powerful feature of this server is that it supports loadable extension modules that enhance its base functionality. Many such modules are available for many purposes. Especially for web application development, an interpreter for one of the "P" languages can be embedded into the web server to enhance the power of the server. Apache HTTPD is considered to be an easy to install and configure web server that needs little attention once it is running.

MySQL

M = MySQL is a widely used open-source relational database management system. It works closely together with PHP as a scripting language, i.e. many websites written in PHP have an underlying MySQL database. PHP is a recursive abbreviation of PHP hypertext preprocessor (originally named "personal home page tool" by its creator, Rasmus Lerdorf, in 1995 [Achour 2006]).

Perl/Python/PHP

P = Perl and *Python* (in addition to *PHP*) are other scripting languages that have been in use since the web began. Early web servers had CGI (common gateway interface) built in. With the help of Perl, Python and PHP, it was possible to use CGI to exchange data between the client's web browser and the server.

When Linux is distributed the other components usually come with it as a bundle. That is why the four components are often used in combination and can be seen as platform. In this sense, the "platform" comprises a web server (Apache HTTPD), a database management system (MySQL) and a scripting language (Perl, Python and/or PHP). The compatibility of these components has grown and been extended over the years.

Wikipedia runs under LAMP

LAMP's combination of a web server, a DBMS and a scripting language suggests that typical e-business systems using web technology and databases can be based on LAMP. A well-known example of a large system on this platform is Wikipedia, the free encyclopedia on the Internet (http://wikipedia.org). It runs under Linux, with content stored in a MySQL database and provided to clients by an Apache HTTPD web server.

LAMP is more limited than the platforms discussed earlier. The combination of the four components basically provides a development and runtime environment for web-based systems. However, since all components are open-source, developers and projects around the world have created a large number of additional components. Among these are content management systems (CMS), application servers and many more.

Tomcat and Geronimo

Application servers that can be used together with the LAMP components have been developed both in ASF (Apache Software Foundation) and other open-source projects. The Tomcat [Apache 2006c] and Geronimo [Apache 2006a] servers, for example, were developed by the Apache Software Foundation, a non-profit corporation in the United States. They add application-server functionality to the web-server functionality of Apache HTTPD. The goal of Geronimo is to provide full application server support for information systems based on Java EE [Apache 2006a].

Similar platforms

Since LAMP has become such a success, similar platforms with slightly different combinations of components emerged, for example, with Windows or BSD Unix as operating system, PostgreSQL as DBMS and IIS as web server. The most prominent acronyms are the following [Jupitermedia 2005]:

LAPP	– Linux, Apache, PostgreSQL, Perl/Python/PHP
WAMP	– Windows, Apache, MySQL, Perl/Python/PHP
MAMP	– Macintosh, Apache, MySQL, Perl/Python/PHP
BAMP	– BSD, Apache, MySQL, Perl/Python/PHP
WIMP	– Windows, IIS, MySQL, Perl/Python/PHP
AMP	– Apache, MySQL, Perl/Python/PHP

The AMP combination has no specific operating system. This indicates that Apache, MySQL and Perl/Python/PHP are actually the important components. The benefits of using them together will show on any operating system. However, Linux is open-source and the other components are often bundled with Linux; therefore this combination is the most popular one.

4 Developing Information Systems

In this and the following chapters, we will discuss different approaches concerning how an organization can obtain its information systems once the decision in favor of a system has been made.

While chapter 7 will deal with buying and introducing software that was developed by others, in particular standard software, the focus in this chapter is on how to build completely new information systems or new modules that extend existing information systems within an organization. By "new modules" we mean that significant development effort is required in order for the project to pass all stages of the software development process.

In contrast to this, limited extensions of a running information system are considered part of the maintenance and support stages. Adding functionality to a new standard software system will be discussed in chapter 7, as part of the customizing process.

The perspective taken in this chapter is that the starting point for the development effort is an approved project proposal (cf. section 2.2.1);

i.e., a managerial level decision to launch a project for building a system inhouse has been made. If no restrictions existed, we could say that the project starts from scratch. In the real world, however, such restrictions often limit the degrees of freedom substantially.

4.1 Starting with a Problem Specification

Constraints that a new information system development effort may need to observe include the following:

Constraints limiting the degrees of freedom

– The existing information systems landscape has to be considered. Most likely the new system will need to be interfaced with the company's ERP system and other information systems.
– The platform on which the new system will run is probably outside the scope of the project. If the company's existing systems are all based on IBM WebSphere, for example, then it is unlikely that a different platform will be chosen for a new system which needs to be integrated.
– Depending on how closely the system is to be connected with existing systems, its architecture may already be predetermined, i.e., it may have to match the architecture of the other systems.
– Many projects have to run under a tight budget and meager staffing, limiting the possibilities of what can be done.

Despite these constraints, the development of new information systems offers a wide array of options and fewer limitations than customization projects.

Project proposal is not operational

Any development project needs a *specification* of the problems to be tackled. The project proposal contains a problem description, but this description is usually just a written text for the approval process, too coarse and not operational enough to identify relevant development tasks. For this purpose, a more formalized high-level specification of the future system is required.

Business process model – a high-level description

Several approaches to define such high-level specifications have emerged in the past. With business processes nowadays being the dominating paradigm for running organizations, this high-level description of the IS needs is usually a description of a business process or a sub-process.

The process specification could simply be a textual description of the major process steps and the resources involved. Since semi-formal specifications have advantages over text, various graphical notations to specify a business process on a high level have come into existence over the years, including the following:

Graphical notations for business process modeling

» Context diagrams in SA (structured analysis) [Yourdon 1989, p. 339]

» Activity and use-case diagrams in UML (unified modeling language)

» Event-driven process chains in ARIS (architecture of integrated information systems) [Scheer 2005]

» Business process diagram in BPMN (business process management notation)

Figure 4-1 High-level business process (example)

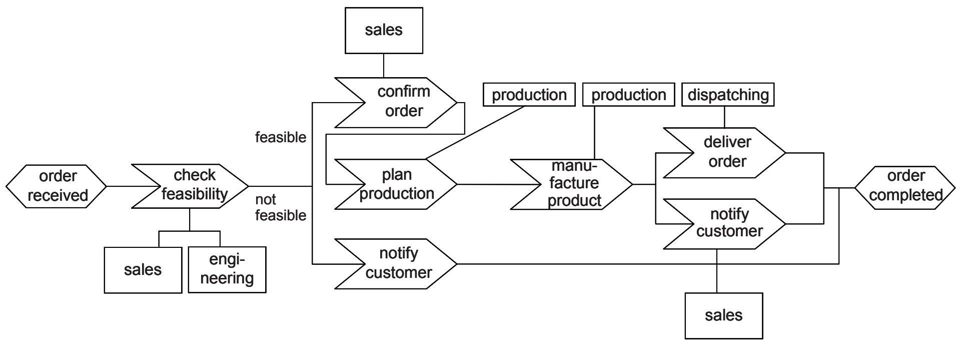


Figure 4-1 shows just one possible way of visualizing a business process. As many notations for business-process modeling exist, the graphical constructs vary. Besides high-level diagrams, all approaches comprise a suite of graphical symbols with appropriate semantics as well as methods and tools for different aspects of modeling and construction. As a process is increasingly refined, more symbols and more meanings are added to the high-level representation. Since methods and tools will be discussed in the following chapter, we do not introduce more notations at this point.

4.2 Process Models and ISD Paradigms

Assuming that an operational problem specification has been created, development of the information system can start. There are many ways to conduct the development effort. Templates arranging development activities into a specified order are called *software process models.* (Note that the term "process" refers here to software development activities and not to business processes as above.) This term can be defined as follows:

Definition: software process model

> A *software process model* is an ordered set of activities with associated results that are conducted in the production and evolution of software. It is an abstract representation of a type of software process.

In a formal view, a software process model can be regarded as a description of a software process at the type level. A particular process is an instantiation of the process model. However, a process model is usually normative ("how things should be done") whereas process instances are actually what happens in reality.

ISD paradigms

A large number of software process models have been proposed since the beginning of software engineering, categorized in many ways, and described by attributes such as:

– linear vs. iterative development,
– sequential vs. incremental development,
– plan-driven vs. agile development,
– model-driven vs. evolutionary development.

Decades of discussion about the best approach to software development have gone by, and method wars have been fought over what might be the best methodology. Most approaches survived, so a variety still exists today. In addition, new organizational forms in the IT industry require new approaches beyond the traditional ones; for example, offshoring and open-source development have to be taken into account.

Approaches to ISD in this book

Out of the variety of old and new approaches, we will discuss the established standard practices from the past, as well as current developments and issues. In the subsequent sections, the following approaches to information systems development will be discussed:

- sequential (waterfall)
- prototyping and evolutionary, RAD
- model-driven
- RUP
- agile
- reuse oriented (web service/orchestration, componentware, COTS)
- offshoring
- open-source

These approaches are not free of overlapping. In fact, most real-world ISD projects have features of more than one category. This means that the software processes in practice rarely follow just exactly one approach but rather include features of other approaches as well.

4.2.1 Sequential Process Model (Waterfall Model)

The sequential process model is based on the idea that the development process can be divided into distinct stages with specified inputs and outputs. The next stage starts when the previous one is completed. Results cascade from one stage downwards to the next stage, just like a waterfall. The flow of work is sequential and basically unidirectional as illustrated in figure 4-2.

First process model in software engineering

The waterfall model was the first process model in software engineering. Its goes back to a systems engineering model that was adapted to software development by Winston Royce [Royce 1970]. Since it was "the" process model for a long time, it has also been called *software life cycle model* (SLC model). Still today, this term is often used to refer to the waterfall model, although many different types of software life cycles have come into existence in the meantime.

Original model has been extended and adapted in many ways

In Royce's original model, seven distinct phases were identified: system requirements, software requirements, analysis, program design, coding, testing and operations. The original model has been extended and adapted by many authors who introduced new phases or arranged phases in a different way, so the number and the names of the phases vary. For example, the design stage is often divided into two stages: preliminary design and detailed design. A widely used version of the waterfall model goes back to Barry Boehm [Boehm 1981, pp. 35-41].

Each stage has a specified result – usually one or more documents that have to be approved before the next stage begins. In principle, the next stage should only start when the result of the previous stage is accepted. For example, the outcome of the requirements analysis stage is a requirements specification as discussed in section 2.2.1. The next stage, design, needs this document as an input. It should not start before the requirements specification is stable and approved.

The fundamental stages of a software life cycle model are illustrated in figure 4-2. The main tasks assigned to the stages are as follows:

Requirements analysis and definition

– *Requirements analysis and definition:* In this stage, the desired functionality of the information system is specified. Requirements of the stakeholders, in particular of the principals and future users, are elicited and analyzed in detail. Requirements analysis is often divided into analysis of:

  » system requirements and
  » software requirements.

  System requirements refer to all components of the information system (i.e. hardware, communication channels, networks, people, organizational units, etc.) whereas *software requirements* address the desired functionality of the software system. The focus in ISD is usually on the latter aspect[§].

  As a result of the analysis process, a software requirements specification is derived. Finding out and describing the requirements correctly and completely is a difficult task, yet crucial for the success of an information systems project. Therefore, *require-*

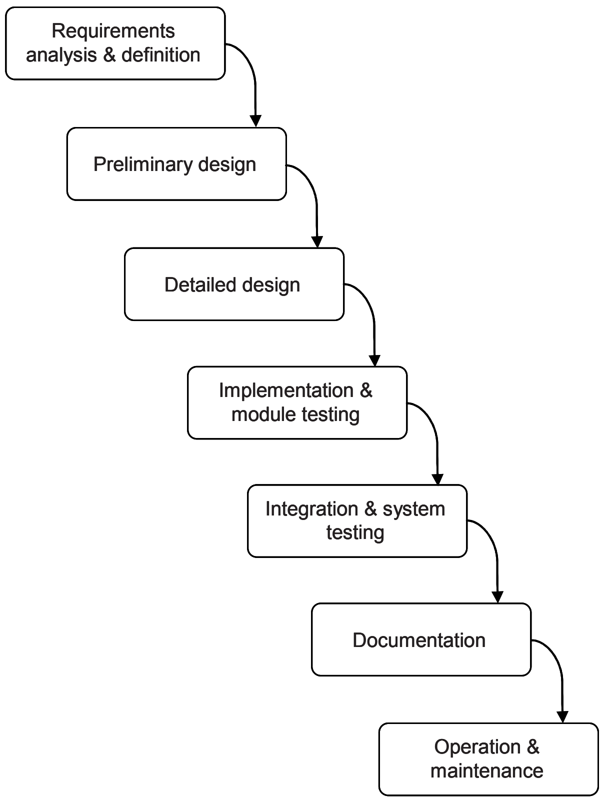
§ The terms "system requirements", "system engineering", and "systems analysis" in non-business contexts usually refer to technical systems where the software is only one of several constituent components, e.g. radar, cruise-control, and telecommunications systems. Electronic, mechanical, electrical, and perhaps other subsystems are equally important as the software coordinating the technical components. Business information systems, on the other hand, focus on information and people. Technical components involved do not carry the same importance as in the before mentioned systems.

ments engineering has emerged as its own discipline. Section 5.1 explains requirements engineering in more detail.

Preliminary design

– *Preliminary design:* The major components of the information system are identified. If an overall system architecture already exists, then the components are placed into this architecture. Otherwise the system architecture is developed first. An architecture is likely to be prescribed if the new system has to work closely together with existing systems.

Figure 4-2 Waterfall model (software lifecycle model)


Requirements analysis & definition
Preliminary design
Detailed design
Implementation & module testing
Integration & system testing
Documentation
Operation & maintenance

For example, if the new system will provide an enterprise service in a larger IS landscape with a service oriented architecture, then the new system has to be designed based on SOA principles.

Detailed design

– *Detailed design:* The major components identified before are specified in more detail and refined into smaller components according to the design paradigm or the particular approach being used in the project. The component interfaces and the interactions between the components are specified. Coarse program logic, workflow and database structures are established.

Implementation and module testing

– *Implementation and module testing:* The system components specified in the design stage are implemented as programs or program modules. Each module is examined through testing if it works properly, i.e. according to its specification. Errors that are detected during testing are removed (debugging). While the original SLC model considered only programs in this stage, further components of the information system also need to be implemented, e.g. the database, forms, reports and workflows.

Integration and system testing

– *Integration and system testing:* The individual components of the system are integrated and tested together to ensure that the entire system works according to the requirements specification. Just as the requirements may include system and software requirements, both the system requirements and the software requirements may have to be validated after integration.

Documentation

– *Documentation:* An important task is documenting the new information system. Depending on the users of the documentation, different types of documents have to be produced. Typical documentations include:

  » end-user documentation (how to use the system),
  » system-administrator documentation (how to run the system),
  » maintenance documentation (how to make changes to the system),
  » API documentation (how to use the application programming interfaces, if provided).

  The last two types of documentation are for developers. Since maintenance programmers usually need to understand the program logic, both the interfaces and the program code have to be described. API documentations basically contain detailed interface specifications. Some information systems may require more types of documentation than the ones listed above.

– *Operation and maintenance:* After the information system has been tested and documented, it is delivered to the internal or external customer. The system is installed in the production environment (i.e. on the computer system it is supposed to run on in practice) and put into operation. It is an empirical observation that from the time the system is put into operation errors are observed and new or changed requirements have to be realized. Since maintenance and operation are overlapping activities, they are usually considered together as one rather long stage.

Operation and maintenance

The drawbacks and advantages of the waterfall model have been extensively discussed for many years. Most authors agree that the assumption of distinct phases performed in strict sequential order does not conform to what happens in practical projects. It is often unrealistic to expect that one phase can be definitely completed with a correct result before the next phase starts.

Drawbacks of the waterfall model

For example, an empirical observation is changing requirements. When a system is built for a customer, quite often the requirements are modified and/or new requirements are formulated by the customer later in the project. This is due to the fact that beforehand, not enough knowledge about the future system existed. Therefore, requirements cannot be specified in detail. As knowledge of the system increases in the course of the project, requirements become clearer and are likely to be adapted at a later stage. In particular, when customers see what they will get (i.e. a running system or at least part of it), requirements may appear in a different light and therefore be redefined.

Changing requirements

Other problems with separating life cycle phases occur during design. Sometimes, specified requirements turn out to be difficult or impossible to transform into a design using reasonable effort, making a revision of the requirements specification necessary. During coding some features of the design may prove difficult to implement unless the design is changed. Likewise, design errors and flaws in the requirements specification are often detected in the implementation stage, requiring repetition of some of the work which was done in earlier stages.

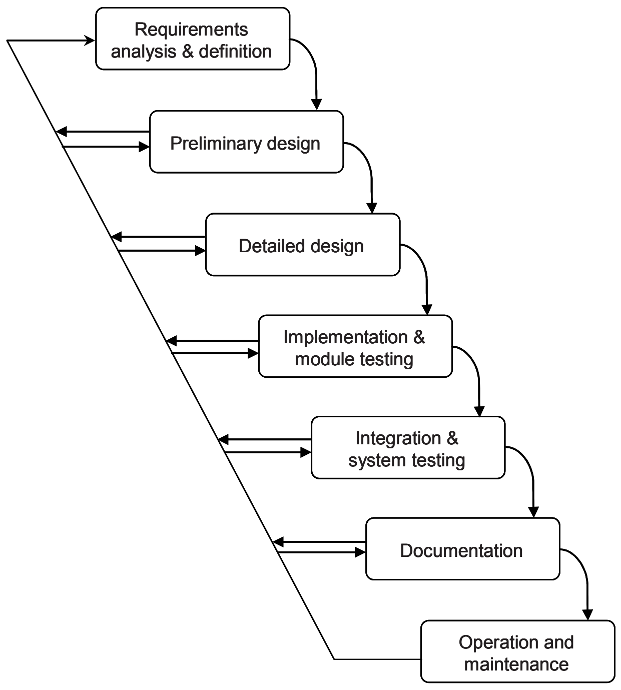
Problematic requirements and/or designs

There are many examples of such situations where work in the next stage has an impact on results of one or more of the previous stages. To cope with these real-world circumstances, modifications of the waterfall model that include revisiting earlier stages were proposed.

Figure 4-3 illustrates the underlying ideas. One immediate variant is that information from the next stage flows back to the previous stage, causing earlier results to be revised. Larger iterations are induced when the need to return to an early stage arises. For example, if contradicting

Information from the next stage flows back requirements are not detected until integration and system testing, then the requirements specification has to be corrected at a rather late point in time, and all the in-between stages have to be executed again. Part of the work that was done before has to be done again. The high cost of improving results of earlier stages late in the process is considered a major drawback of process models which are based on the waterfall model.

Figure 4-3 Software life cycle model with iterations


Requirements analysis & definition
Preliminary design
Detailed design
Implementation & module testing
Integration & system testing
Documentation
Operation and maintenance

Waterfall model has strong advocates

Although the waterfall model and its extensions are often discussed in a negative undertone, there are strong advocates of these models. Large organizations have established versions of the model for their own projects because this model provides a structured approach which is easy to understand and to use in communication between development personnel and managers. Having distinct phases also means that milestones and deliverables can immediately be attached to the phases, providing crisp points in the project for management control and action. Contracts with external software firms can be based on deliverables that are a result of a stage in the waterfall model.

Advantages

Putting emphasis on completing early stages such as requirements analysis and design before proceeding further makes sense. Having a complete and consistent set of requirements before the design is created and the software is coded helps to save money. If analysis-and-design mistakes are detected in the coding and testing stage, then a lot of the earlier work has to be re-done.

Appropriate when requirements are clear

The software life cycle model is considered useful for large projects where reliable requirements can be specified in advance. This is the case, for example, when the problem domain is well-known, when the project team has experience with similar IS development projects and when customers are not directly involved in the project (e.g. developing shrink-wrapped standard software).

4.2.2 Evolutionary Process Models and Prototyping

A new way of thinking

The obvious drawbacks of the sequential approach stimulated a different approach to thinking about reasonable software processes. This new way of thinking already began in the 1970s and was very strong in the 1980s. In fact, Winston Royce in his often cited 1970 article had not actually advocated the waterfall model but pointed out its shortcomings. Consequently he proposed an iterative approach similar to the one illustrated in figure 4-3 [Royce 1970, p. 9].

SLC iterations essentially correct flaws

The fundamental disadvantage of the waterfall model and its extensions is the sequential flow of information and results from one stage to the next. Even in its iterative variants, the main process is a sequential one. Iterations essentially correct flaws and improve specification and design features that were badly done before, either because of a lack of knowledge or because of mistakes.

Abstract specifications are difficult

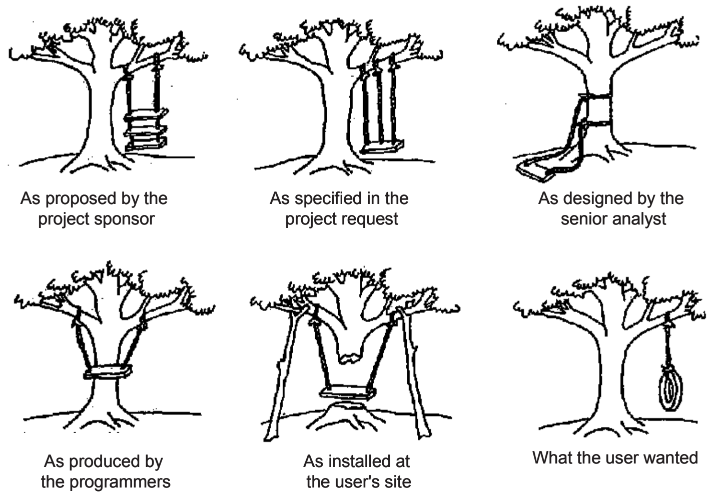
We pointed out before the difficulties to establish correct, complete and consistent requirements and to design system components on an abstract level without a line of code written. These difficulties lead to different approaches that intermingle analysis, design and implementation of an information system and let users see early what the final system will be like.

An often displayed cartoon in the software engineering literature illustrates the dilemma of the waterfall model (see figure 4-4): A long period of time goes by between when a project's requirements specification is produced and when the customer sees the final product. If the requirements specification did not mirror exactly what the customer actually wanted, this divergence will only show when the product is delivered – and much money has already been spent.

To overcome these drawbacks of the waterfall model, two guiding principles that are fundamentally different from the sequential approach were established:

1. Making software development an evolutionary process
2. Building prototypes

Figure 4-4 "What the user wanted"


As proposed by the project sponsor
As specified in the project request
As designed by the senior analyst
As produced by the programmers
As installed at the user's site
What the user wanted

While most systems evolve after they have been installed (i.e. they grow and change), *evolutionary development* means that system growth and changes are already embedded as integral parts in the development process. Iterations in the software lifecycle model serve this purpose to some extent, but they are considered a necessary evil rather than a welcome process feature.

IS comes into existence in an evolutionary process

In a truly evolutionary process model, the premise is that the information system comes into existence through evolution: Starting with incomplete and perhaps insufficient knowledge about what the final information system has to be like, a limited subsystem is created in the beginning. Continuing from this subsystem, an enhanced, extended, and/or better subsystem is created. This subsystem may include new or better components than the previous subsystem. The process continues until a satisfactory and complete information system has evolved.

Evolutionary development has various incarnations. Many authors have proposed approaches that create a sequence of running subsystems until the final system is established. The most important ones are:

Iterative enhancement

– *Iterative enhancement:* Requirements are specified as objectives in the beginning. Some requirements are then selected and realized in three phases: design, implementation and analysis. The goal of the analysis phase is to evaluate the subsystem created and perhaps modify the design. In the next step, the subsystem is extended by selecting and realizing more requirements etc. The process ends when all requirements have been dealt with [Basili 1975]. Whereas design changes and extensions are part of the methodology, major changes of the requirements are not. This means that iterative enhancement is suited for projects where the requirements are more or less stable from the beginning onwards.

Incremental development

– *Incremental development:* The overall system is developed as a sequence of increments [Mills 1980]. Customers set priorities regarding their requirements for the system. Subsystems are identified that realize a subset of the requirements. After the overall system architecture has been designed, a subsystem fulfilling the most important requirements is designed in detail, implemented, tested, delivered and evaluated by the developers and the users. This might lead to new insights about the requirements that are taken into consideration when the next increment is developed. This increment undergoes the same development process. After its integration with the previous increment(s), the new system is evaluated, and the next increment is developed, etc.

Advantages and disadvantages of incremental development

Advantages of incremental development include that the customer gets a running system after a shorter time and does not have to wait until the entire system is completed. The shortcomings of the initial increments can be avoided in later increments, so the overall system quality is enhanced.

The major *disadvantage* is that it is not easy to split up a system into distinct subsystems if no overall design is made. Services or modules needed by all subsystems are difficult to identify beforehand. In addition, if the increments are developed in completely separate subprojects, there is a risk that the subsystems will be heterogeneous and not behave in the same way, which would adversely affect the user-friendliness of the system.

Versioning

– *Versioning:* The final system evolves from a sequence of preceding versions. While there is no sharp distinction from iterative enhancement and incremental development, versioning implies that more or less the same functionality is available in the successive versions. Yet the next version represents a better implementation than the previous one, be it through an enhanced architecture or design, through better coding or through improved requirements.

Requirements are part of the evolution

Taking the most severe problem of the waterfall model into account – i.e. capturing the requirements in a correct, complete, and consistent manner – we believe that the most important progress when using an evolutionary approach, as opposed to a sequential one, is the embedding of the requirements definition into the evolution loops. This means that the requirements document is not established only once and for the entire project, at an early point in time when many things about the new system are not clear yet, but that requirements can be refined and revised throughout the project.

Ease of change is required

This is certainly not compatible with a sequential process model in which all subsequent stages depend entirely on a definite requirements specification. Consequently, the design and implementation stages must be flexible in the sense that a system design which has only been established once may need to be changed, and the code written for the old design should be easy to modify in the next loop. Ease of change is a (non-functional) requirement that can be substantially supported by automated tools.

Taking the evolution of the requirements specification as well as the underlying ideas of iterative enhancement, incremental development and versioning into consideration, a generic process model for evolutionary information systems development can be described through the following steps:

A generic process model

1. Start with an analysis of the problem and an provisional set of requirements for the project.
2. Create a preliminary specification of the requirements as input to the first development cycle.

3. Establish a preliminary system design based on the current requirements specification.
4. Select those requirements or functionalities that should be actualized next.
5. Develop a detailed design of the system version or the subsystem defined in the previous step.
6. Implement, test and debug the current system version or subsystem.
7. Evaluate the current system version or subsystem (involve both developers and customer/users in the evaluation).
8. Continue with step 4 until a satisfactory solution to the problem is obtained; i.e., revise the requirements specification and subsequently the design based on the evaluation.

Figure 4-5 summarizes the process model. To keep the figure simple, not all possible branches have been explicitly indicated. For example, an intermediate version or a functioning subsystem providing value to the customer may be installed and implemented before the complete system is available, if that subsystem can run by itself. In addition, a bypass of the design stage is possible because design revisions are only necessary if modified requirements call for a design change.

Several development cycles

The system undergoes several development cycles in which the requirements are continuously reformulated, refined and/or improved. The loop comes to an end when all development objectives are fulfilled. By objectives we mean 1) the requirements stated, 2) the desired software quality, and 3) a satisfactory solution to the actual problem that initially caused the customer to start the project. Three to four iterations are typically observed in mid-size practical projects.

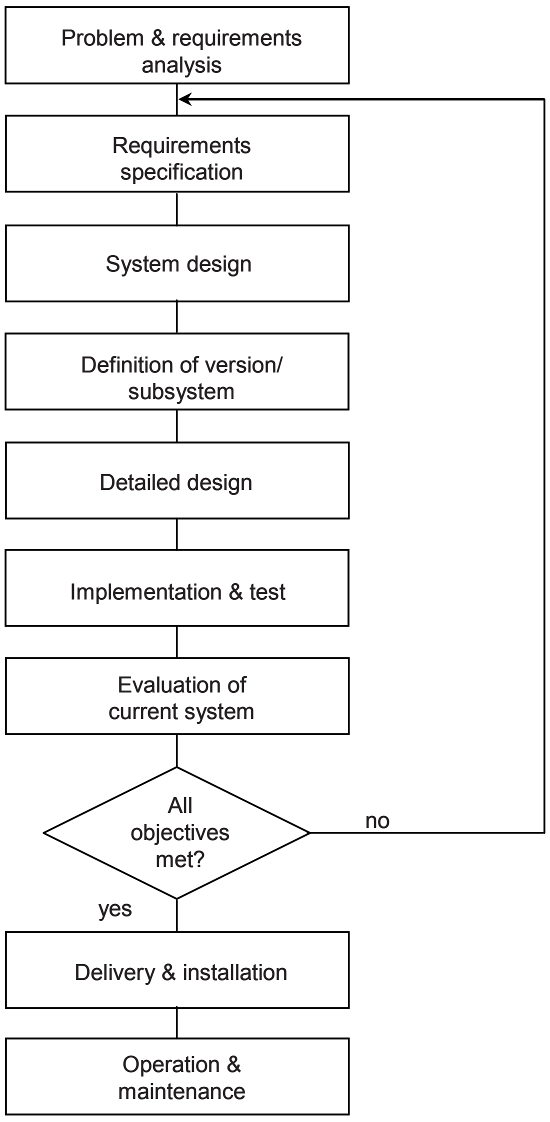
Disadvantages of evolutionary development

Although an evolutionary process model has many advantages over a sequential process model, *disadvantages* should also be noted. When a system is intended to evolve continuously, it is difficult to establish instruments for management control such as milestones and deliverables. Since the result of a development cycle is based on a preceding evaluation and the result of the previous cycle, it is not possible to predict what will be the result of the next cycle and when that result will be available. Likewise, contracts with external providers are difficult to formulate if crisp deliverables cannot be specified.

Structure suffers from "patchwork"

Furthermore, as the system undergoes continuous change, the final system structure may degenerate into an architectural patchwork, making maintenance and future extensions costly.

Figure 4-5 Evolutionary process model



Tool dependencies

As mentioned before, automated tools can significantly enhance development productivity. Since change is inherent to evolutionary development, tools that generate code are particularly helpful. On the other hand, the pressure to use such tools creates tool dependencies which may be counter-productive in the long-run. For example, a different toolset may turn out to be better suited for the next cycle. However, it will be difficult to adopt the new tool if system components meant to remain in the next version are bound to the old toolset.

Prototyping

Prototyping is an established engineering practice

Building system prototypes is a well-established practice in engineering. When a new product is under development, working prototypes are built first in order to study design and manufacturing options or customer acceptance. Prototypes in information systems development serve similar purposes. IS prototypes are running subsystems that help developers or users to gain insights into the future system. These insights would not be available if only abstract paper documents describing the system were created.

Prototyping in information systems development can be used in two fundamentally different ways:

– Throw-away prototyping
– Evolutionary prototyping

Throw-away prototyping

The main purpose of a *throw-away prototype* is to provide an object of study to developers or users that they can explore and gain experiences with. The prototype is only used for that purpose and later discarded (cf. figure 4-6). Since it does not constitute a part of the final system that will be delivered to the customer, throw-away prototypes are often created in a "quick and dirty" manner; i.e. software quality is not given high priority.

Powerful software tools are needed

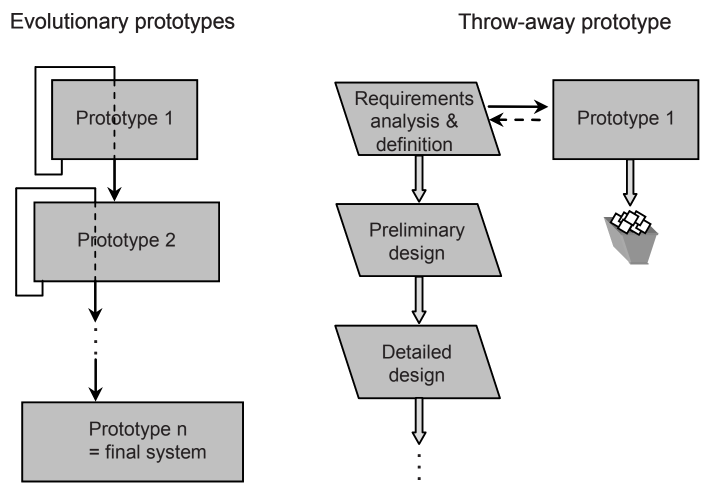
Powerful tools are needed to create prototypes fast. Although a prototype could be written in a conventional programming language, the speed factor calls for high-level tools. Such tools can aid a quick creation of a prototype through drag-and-drop features and code generation, which requires only little hand-coding. In particular, the creation of graphical user interfaces should be facilitated by tools.

The term *rapid prototyping* has also been coined for this approach. Rapid development does not necessarily imply that the prototype will be thrown away afterwards.

Evolutionary prototyping

Evolutionary prototyping means that an initial prototype will be absorbed into the next one and so on, as illustrated in figure 4-6. The final system will contain code that has evolved from the previous prototypes. This process was discussed above under the topic evolutionary process models. The project will follow such a process model when prototype evolution is the approach of choice.

Figure 4-6 Throw-away vs. evolutionary prototyping



As opposed to evolutionary prototyping, throw-away prototypes are mostly used *within* a software life cycle stage, to support core activities of this stage. From this viewpoint, prototyping variants include the following:

Requirements prototyping

– *Requirements prototyping* as part of the requirements analysis and definition stage is the most widely used form of prototyping. The aim here is to assist the project staff to elicit and to validate user requirements. Prototypes are built to give the customer and/or the users an idea of what the implementation of their requirements will be like. A prototype helps them to make vague or fuzzy requirements explicit so that they can be mapped on to information system functionality. Section 5.1 discusses requirements engineering in more detail.

User-interface prototyping

– *User-interface prototyping* is similar to requirements prototyping in that those parts of the system representing the system's interface to the user, i.e. the graphical user interface (GUI), are developed quickly. The purpose of user-interface prototyping is to demonstrate the look-and-feel of the future system to the customer and/or the users. The prototype is likely to be a mock-up; i.e. functionality behind the GUI is not implemented yet. A powerful GUI builder tool such as Visual Studio .NET (for Windows based systems) or JBuilder (for Java based systems) is essential for user-interface prototyping.

Design prototyping

– *Design prototyping* is an approach supporting the design stage(s). It is different from the previous ones in that it targets the developers and not the users. Design decisions can affect the ease or difficulty of implementation and maintenance and thus effect future costs positively or negatively. If design consequences are not immediately obvious, building prototypes to try out one or more design options helps to assess implications of the options. In this case, the prototype will not contain a sophisticated GUI but rather a complete implementation of a small part of the overall functionality.

The *advantages* of prototyping are manyfold. Evolutionary prototyping is a means to accelerate delivery of the system to the customer. Stable subsystems can be installed for practical use before the entire system is completed. In today's fast-changing business environment, the speed of delivery can be a critical factor with regard to the competitiveness of an enterprise.

Prototyping makes things clear early

Prototyping helps to make things clear early. Customers and/or end-users see at an early stage of the project what the final system will be like. In particular, a prototype can help to derive and formulate requirements for the subsequent development phases. To both developers and customers, the size of the final system becomes more transparent, facilitating time and cost estimation. Design prototypes can provide helpful insights regarding the expected effort. In fact, prototypes are sometimes built especially to support cost estimation.

Disadvantages of prototyping

Obvious *disadvantages* on the other hand have prevented many organizations from adopting prototyping, in particular evolutionary prototyping, as a general approach to information systems development. Some of these problems we discussed above, in the context of evolutionary process models, such as management and contractual problems, software quality and tool dependencies. Further problems related with the use of prototypes have been identified as the

– "normative force of facts",
– pressure to release,

– uncontrollable growth of requirements,
– lack of project management methodology.

"Normative force of facts"

➡ *"Normative force of facts":* The rationale of prototyping is to enable exploration and gaining of experience so that the solution or the next prototype will be better than the current one. In contrast to this, developers tend to "save" their work and reuse it in subsequent versions. Inappropriate solutions will survive in this way. Likewise, the perception of users may be prejudiced by the current prototype. As they are not aware of alternative solutions to the problem, the exemplary implementation is considered "the" solution, no matter if a better solution exists or not. Tool restrictions and peculiarities may further narrow the solution space. Due to these factors, the example (i.e. the prototype) determines the final product, which is not the idea of prototyping.

Pressure to release

➡ *Pressure to release:* Customers, end-users and managers are usually not IT experts. When they see a running prototype, they may not be aware that it is still a long way to go from the prototype to a stable, robust and efficient production system ("it works, why can't we release it?"). It may be difficult to explain to non-experts why still three times the already elapsed time will be needed until the final system is available, and that ignoring software quality now will increase maintenance costs later.

Uncontrollable growth of requirements

➡ *Uncontrollable growth of requirements:* A fundamental assumption in most forms of prototyping is that customers and/or end-users are involved in the development process. As the project progresses, things become clearer and the state of knowledge about the problem and its solution increases. In many cases this leads not only to requirements changes but also to new requirements, implying more work for the development team. Negative consequences can be: frustration on the developers' side because of a higher workload and having to throw away results of their previous work; difficulty staying on schedule and budget; and delay of delivery, the latter leading to customer dissatisfaction. Project managers have to keep an eye on balancing growing user requirements with the project schedule and budget. If changes have not been provided for in the initial agreement, conditions and terms may need to be renegotiated.

Lack of project management methodology

➡ *Lack of project management methodology*: While the waterfall model is accompanied by a widely used set of project management methods, such a common and accepted methodology does not exist for prototyping. In many organizations, evolutionary prototyping is considered inappropriate for practical projects. A major reason for this is that an evolutionary process model does not provide crisply defined points for management control and action as a sequential process model does. It should be noted, however, that project management methodologies for evolutionary prototyping *are* available. They are just not as common as the standard SLC based methodologies. (This author already developed a project management methodology for evolutionary prototyping in 1987 [Kurbel 1987, Kurbel 1990].)

4.2.3 Model-driven Information Systems Development

Models play an important role in information systems development. They can be used in various phases of the development cycle. For example, the entity-relationship model (ERM) is often employed to capture the essential data for an information system and their interrelations as input for the creation of a database.

Models were for documentation

However, consistent use of models throughout the development stages was not a common approach in practice; rather models were created subsequently on paper for the documentation, when the development was already finished. Nowadays modeling is supported by powerful tools that will be discussed in detail in chapter 5.

mdISD is business oriented

Model-driven information systems development (mdISD) has a stronger orientation towards business information systems than the previously discussed process models[§]. While these models are rather general, targeting technical software in engineering as well (or in the first place), mdISD supports primarily business problems. There are two fundamental ideas on which mdISD relies:

1. the use of models at all stages of development,
2. the automated transformation of models into code or other models.

While automatic code generation from formal specifications had been a dream of computer scientists since the early times of programming, this dream largely did not come true. Only when semi-formal methods and diagramming techniques matured and the automated tools

§ We use the term "model-driven information systems development" (mdISD) instead of other terms such as "model-driven development/model-driven architecture" (MDD/MDA) to express the specific focus on (business) information systems development.

supporting these techniques became available in the early 1990s, did model-based information systems development become a serious approach for large-scale practical projects.

Information engineering (IE)

James Martin – a pioneer of mdISD

A pioneer in the field of model-driven information systems development was James Martin who established the discipline of *information engineering (IE)*. This was a comprehensive approach to enterprise-wide modeling of all aspects of information systems and transforming the models into running systems. Martin defined information engineering as:

Definition: information engineering

> "The application of an interlocking set of formal techniques for the planning, analysis, design, and construction of information systems on an enterprise-wide basis or across a major sector of the enterprise." [Martin 1989, p. 1]

IE is supported by "an interlocking set of automated techniques in which enterprise models, data models, and process models are built up in a comprehensive knowledge base and are used to create and maintain data processing systems." [Martin 1989, p. 1]

Focus on enterprise-wide information processing

The comprehensive information-engineering view covers all stages of IS planning and development, starting from strategic planning down to technical construction of programs and databases. The focus of IE is not on a single information system but on enterprise-wide information processing as a whole. Separate views of information systems are integrated. According to the state-of-the-art at that time, these views are data, functions and processes – all analyzed and modeled within a common framework.

Information engineering consists of four main stages, as illustrated by the pyramid view in figure 4-7:

ISP (information strategy planning)

– *ISP (information strategy planning)* is the top stage where strategic goals, critical success factors and information requirements of all major parts of the enterprise are determined. The result of information strategy planning is a global model of the enterprise and its division into business areas.

BAA (business area analysis)

– On the second level, *BAA (business area analysis)* is performed within one or more major sectors of the enterprise. Data models (e.g. entity-relationship diagrams), process models (e.g. decomposition diagrams) and other models are developed here, and desirable information systems within the business areas are defined.

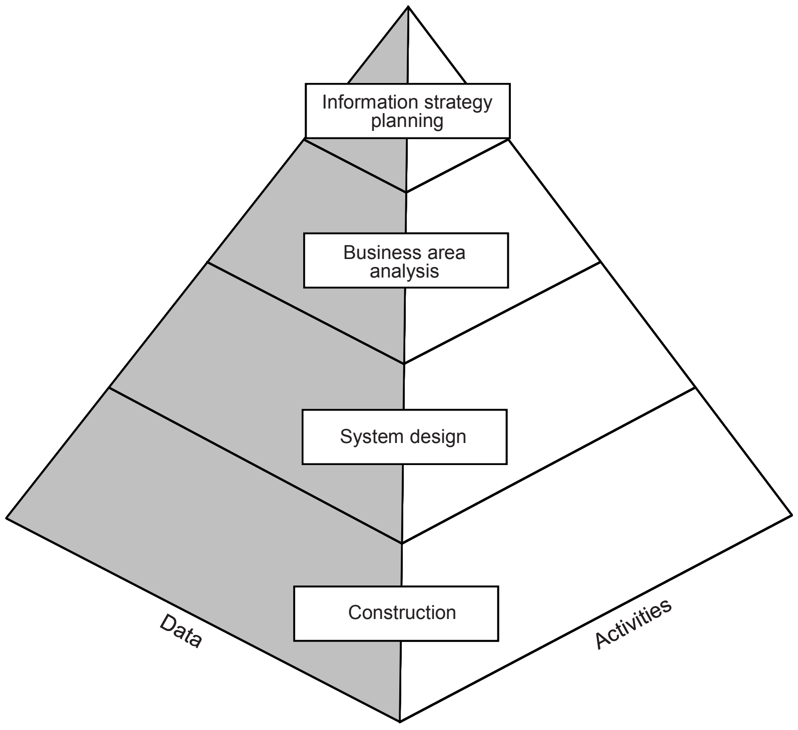
- *SD (system design)* is the third stage where procedures, data structures, screen layouts, windows, reports etc. are specified.

SD (system design)

- In the fourth stage, *construction*, programs and data structures are implemented, tested and integrated.

Construction

Figure 4-7 Information engineering pyramid [Martin 1989, p. 4]



Although the idea of capturing and modeling an entire organization's information systems needs is independent of tools, turning this impressive approach into running systems is not possible without powerful tools at all stages. A major objective of IE is to generate code from models automatically. In fact, James Martin with his IE approach was a major promoter of I-CASE (integrated CASE), because information

IE needs powerful CASE tools

engineering without integrated CASE tools was just not possible[§]. Well-known toolsets supporting IE in the 1990s were ADW (Application Development Workbench) and IEF (Information Engineering Facility) [Stone 1993].

IE encyclopedia

Model and tool integration as well as code generation require a central storage place or repository. In IE it is called the *encyclopedia*. All the information collected during the stages of information engineering is transformed into a common representation format and stored in the encyclopedia.

"Knowledge coordinator"

One of the heavily used figures in IE is the so-called *knowledge coordinator,* an avatar that has the encyclopedia in its head. Figure 4-8 shows this view of the encyclopedia. (It should be noted that the meaning of the term "knowledge" is not the same as in artificial intelligence. A more appropriate term for the IE artifacts would be "information.")

The comprehensive information engineering approach was very popular among business informatics academics because it satisfied a major requirement – integration of processes, functions and data – that had been demanded in theory for a long time. In practice, IE unfortunately did not gain the same level of acceptance for two reasons.

Sufficiently powerful CASE tools were not available

Firstly, extremely powerful CASE tools were needed. Tool manufacturers were not capable of providing such efficient tools that would satisfy the heavy demands posed by the theoretical concepts. One of the largest failures in software engineering was IBM's repository manager project within AD/Cycle. AD/Cycle was an equally comprehensive approach as information engineering, based on a repository (the term "repository" was actually coined by IBM in that project). After more than ten years of development effort and hundreds of person-years spent, the repository manager was finally withdrawn by IBM. The problem was just too difficult, and it turned out to be impossible to create a repository that could work with acceptable performance.

Cumbersome code generation

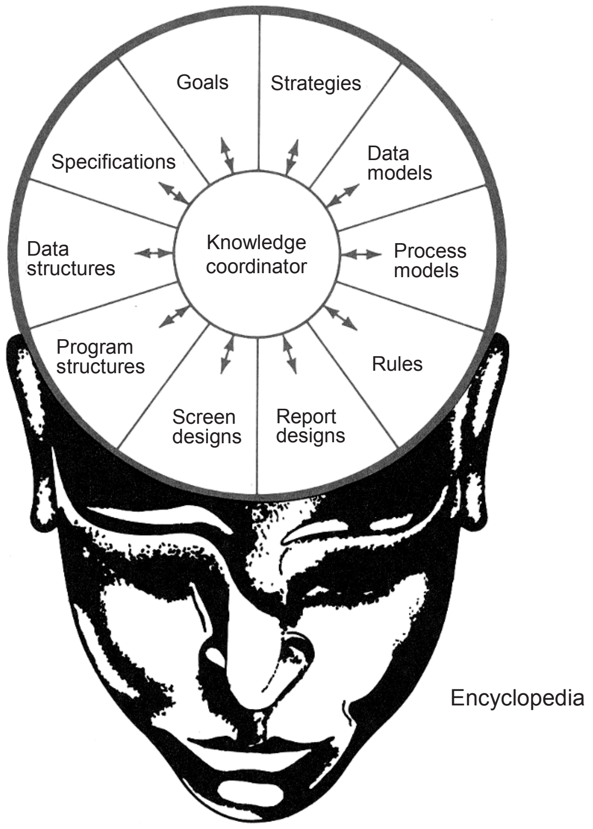
Secondly, code generation from the activities-oriented models was cumbersome. IE tools in general were nicely integrated, able to exchange information and adapt representation formats smoothly. Generating database structures from data models was fairly straightforward, but generating program code was a rather awkward and error-prone procedure. A lot of manual work not much different from ordinary 3GL programming had to be done to make code generation really work.

Thirdly, the dissemination of standard business software in the late 1990s, especially for enterprise resource planning and related areas, made enterprise-wide information modeling somewhat redundant. The

§ CASE tools will be discussed in section 5.3.

purpose of modeling is to finally obtain running information systems. When information systems for ERP, SCM, CRM etc. are already there, then the modeling was performed by the software vendors before and there is no point in doing it once more.

Figure 4-8 Repository in information engineering [Martin 1989, p. 15]



Model-driven architecture (MDA)

OMG introduced MDA

The model-based approach was revived in the beginning of the 21st century through the concept of a *model-driven architecture (MDA)* introduced by the Object Management Group (OMG). The aim of MDA is to separate business and application logic from underlying platform technology. A complete MDA specification consists of a definitive platform-independent base model, plus one or more platform-specific models (PSM) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform [OMG 2006]. Application systems based on MDA consist of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that has to be supported.

Functionality and behavior are modeled only once

According to the OMG, a major advantage of a model-driven architecture is that "... it is not necessary to repeat the process of defining an application or system's functionality and behavior each time a new technology (web services, for example) comes along. Other architectures are generally tied to a particular technology. With MDA, functionality and behavior are modeled once and only once. Mapping from a PIM through a PSM to the supported MDA platforms is being implemented by tools, easing the task of supporting new or different technologies" [OMG 2006].

50% to 70% of PIM-to-PSM transformation automated

Having observed the difficulties of IE tool providers with automated model transformations, we do not completely share the optimism expressed in the last sentence. Indeed, the OMG admits that today's tools typically automate only 50% to 70% of the PIM-to-PSM transformation, leaving the rest to be manually coded and adapted [OMG 2006]. However, automation of the PSM-to-code transformation is said to be near 100%.

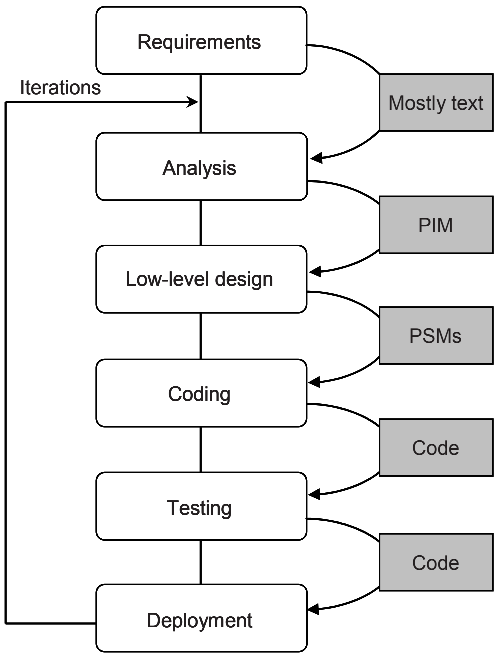
Outputs and inputs of stages are models

Along with the MDA, a development life cycle model was introduced. It is not completely different from other software life cycle models, because it consists of the same or similar stages as other models. However, the outputs and inputs of some stages differ in that *models* are explicitly defined as the results that connect stages. As figure 4-9 shows, the result of the analysis stage is a platform-independent model (PIM). This model is input into the design stage in which a platform-specific model (PSM) is developed, or more of such models if more than one platform is targeted. Coding and testing is, of course, platform specific. Iterations are part of the MDA life cycle model.

Methods and tools for MDA are closely connected with UML. Although not formally required, UML is a key enabling technology for the model-driven architecture and the basis of 99% of MDA development projects [OMG 2006].

UML is an enabler for MDA

Figure 4-9 MDA software life cycle [Kleppe 2003, p. 17]



ARIS (architecture of integrated information systems)

A working example of an mdISD approach is *ARIS (architecture of integrated information systems).* ARIS started in the beginning of the 1990s as an approach to enable the development of integrated information systems. It was created by a distinguished business informatics professor, August-Wilhelm Scheer, and established as a suite of commercial tools by IDS Scheer AG in Saarbruecken, Germany. Similar to information engineering, several views of information systems were considered: data, functions, organization and control [Scheer 2000]. In

Created by business informatics professor A.-W. Scheer contrast to IE, ARIS did not require an enterprise-wide top-down approach across the whole enterprise; it was suited for individual information systems as well.

All of the mentioned views were supported by specific methods. With the exception of event-controlled process chains, most methods in ARIS were not new. Instead, established methods were used and arranged in a comprehensive framework, for example entity-relationship modeling, decomposition diagrams, organizational charts and sequence diagrams.

Event-controlled process chain (EPC)

The focus of ARIS later shifted more towards business process modeling (BPM) and tooling support for this purpose. Based on the concept of event-controlled process chains, a set of tools for BPM was developed. The core components of an *event-controlled process chain (EPC)* are events and functions (process steps). Events trigger functions and the execution of a function usually terminates in an event that may trigger another one or more functions.

Market leader in business process modeling

Today, ARIS is a market leader and in Europe considered a quasi-standard for business process modeling. SAP users often apply ARIS to model their processes before customizing the SAP systems they are going to implement in their organization. The ARIS toolset provides tools all the way down to the construction stage, supporting code generation to a significant extent. The underlying methods are either based on UML or developed specifically for ARIS.

ARIS abstraction levels

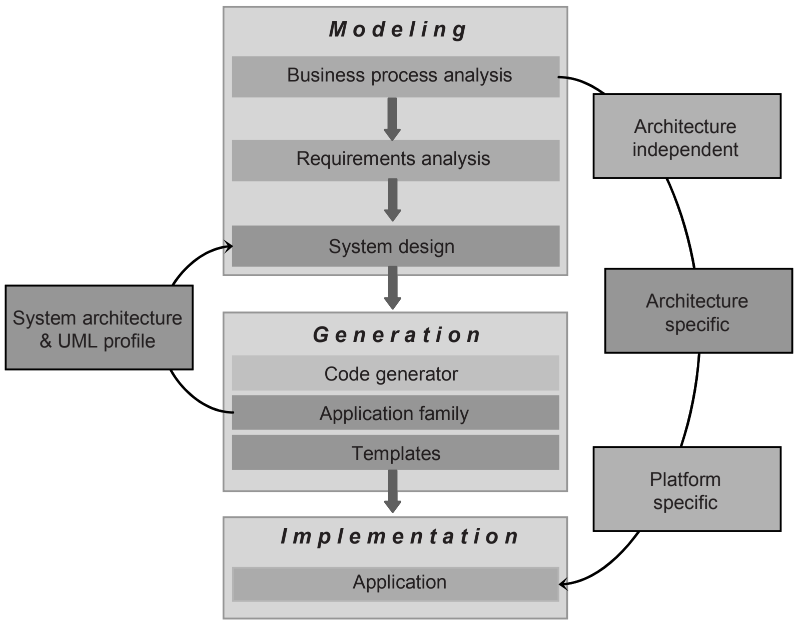
The initial process model for information systems development in ARIS contained simply the following stages: business problem, requirements definition, design specification and implementation. These stages corresponded to what was called "levels of description", i.e. abstraction levels for describing the problem, the requirements, the design and the implementation. Models were created on the problem, the requirements and the design levels. Abstraction levels mapped quite easily onto development stages, because obviously models of the business problem are created first, then requirement models, then design models, finally followed by the implementation in code.

Modeling, generation and implementation

With increasing focus on business process modeling and tools, the process model for ARIS nowadays contains three major levels as illustrated in figure 4-10: modeling, generation and implementation. Models created in the business-process-analysis and requirements-analysis stages are independent of both specific architectures and specific platforms. System design is done for a particular architecture, and the generated code is obviously platform-dependent.

In the business-process-analysis stage, business processes are primarily modeled with the help of event-controlled process chains. For requirements analysis, UML modeling techniques are provided. In order to derive a UML based model of requirements from EPC based results of business process analysis, the latter ones have to be mapped to UML constructs, e.g. use cases (cf. section 5.1.3).

Figure 4-10 ARIS model-driven process model [Andres 2006, p. 3]



For code generation later on, certain parameters regarding the selected type of information system (application family) must already be provided in the system-design stage. The code generator can then, with the help of more information about the application family, convert the design model into source code and other artifacts.

Templates of the application family determine what code is actually generated from the design [Andres 2006]. The final implementation usually requires manual completion and adaption of the generated code.

4.3 Rational Unified Process (RUP)

Ivar Jacobson, Grady Booch and James Rumbaugh

Good and bad experiences from other process models were taken into consideration when RUP was born. RUP was created in a joint effort by three well-known experts in object-oriented analysis and design, Ivar Jacobson, Grady Booch and James Rumbaugh. Each of them had established a proven object-oriented methodology by the time they joined Rational Software Corp. to create a unified approach based on the three parallel predecessors. The major outcomes of this effort are *RUP (Rational unified process)* and *UML (unified modeling language).*

RUP: process model, framework and methodology

RUP is a process model, a framework to create process models, and a methodology to develop software systems. As a *process model*, it supports incremental development, dividing large projects into smaller subprojects. Characteristics are iterations and increments, strong involvement of all stakeholders (developers, architects, end-users, managers, customers etc.) at all stages, and built-in quality assurance. Although some authors say that RUP is requirements-driven, others state explicitly that it is not. It is a matter of perspective. Requirements certainly play an important role; yet they are not defined just once and for all times at the beginning, but evolve during the process [Kruchten 1996, p. 14].

The process model can be used "as is", but usually it is adapted to the organization's or the project's specific needs and characteristics. That is why RUP is also an adaptable *framework* for deriving and tailoring specific process models. Being closely connected with UML, RUP provides a *methodology* for developing software systems that is supported by various UML constructs.

RUP phases

The process model is two-dimensional. The dimensions are phases and disciplines (originally called workflows). Phases extend in time, and disciplines expand into activities (cf. figure 4-11). The *phases* are called:

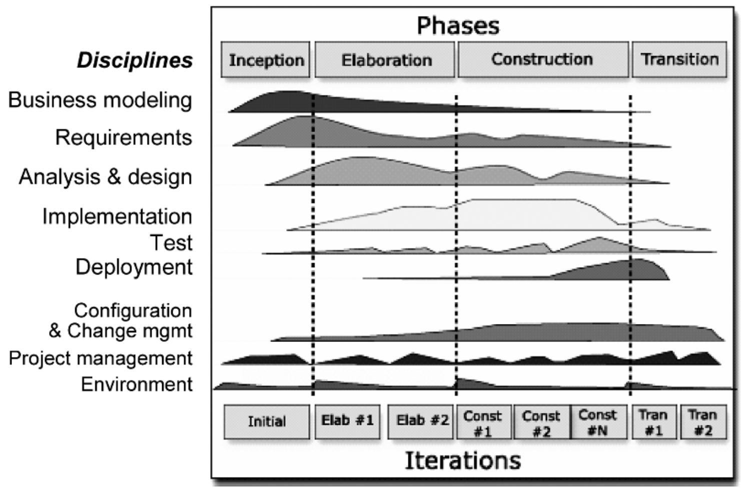
» Inception
» Elaboration
» Construction
» Transition

Workflows were originally distinguished into core process workflows and core supporting workflows. Since the renaming of workflows to *disciplines* in 2001, a further subdivision is not common any more. Thus the disciplines are:

RUP disciplines

- » Business modeling
- » Requirements
- » Analysis and design
- » Implementation
- » Test
- » Deployment
- » Configuration and change management
- » Project management
- » Environment

Figure 4-11 RUP lifecycle – phases and disciplines§



The hump-chart diagram in figure 4-11 is one of the landmarks of RUP. It expresses several things quite clearly:

§ Ambler 2005a, p. 2.

Disciplines extend across phases

1. Disciplines extend across phases. This means that typical activities such as modeling, analysis, design, implementation and test are not confined to one phase but are ongoing activities during the entire life cycle.
2. The humps in the curves indicate how much effort will be needed at what times. For example, most of the analysis and design effort occurs in the elaboration phase, whereas the implementation curve has its highest point in construction.

Iterations occur within phases

3. Iterations occur within phases. Examples of iterations are given at the bottom of figure 4-11: Inception has only one iteration in this example; the elaboration, construction and transition phases consist of two, three and two iterations, respectively.

"Serial in the large, iterative in the small"

Iterations are intended for all phases, yet only within a phase. The phases as such are sequential. For example, the construction phase starts when elaboration is over. However, activities within a phase (e.g. analysis, design, implementation and test) may be performed repeatedly until a satisfactory outcome is obtained. In this way, RUP combines sequential and iterative process aspects in one process model ("serial in the large, iterative in the small" [Ambler 2005a, p. 1]).

Hump chart shows the workload

The hump chart shows how typical projects behave regarding the workload of disciplines and phases. Although all projects are different, they exhibit similar distributions. It should be noted, however, that effort and time (schedule) are not identically distributed. Typical distributions are outlined in figure 4-12. The reason why effort and schedule have slightly different numbers is because the manpower utilization which is underlying the effort criterion usually varies across phases.

RUP development cycle and evolution

A complete pass through the four phases is called a *development cycle*. Since RUP is based on the rationale that a software system evolves over time, the process does not end with the "final" release delivered to the customer. New environmental factors, requirements or technologies may call for substantial extensions or modifications.

Figure 4-12 Typical effort and time allocation [West 2003, p. 4]

Phase	Inception	Elaboration	Construction	Transition
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

This means that the cycle starts over again: inception → elaboration → construction → transition. Since an existing software system is already available, the new cycle does not need to begin from scratch. Thus the inception phase may be considerably shorter or even omitted. In the latter case, the first full phase of the new cycle would be elaboration.

Figure 4-13 Evolution cycles [Ambler 2005a, p. 12]

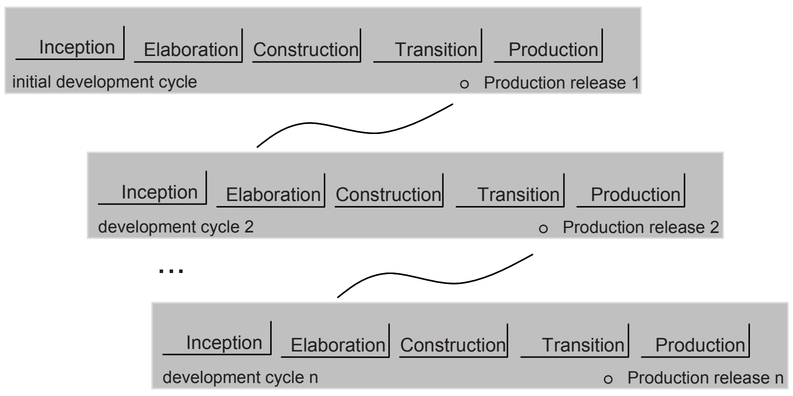


Figure 4-13 shows the evolution of a software system through its cycles. New cycles may become necessary throughout the lifetime of the system, as long as the stakeholders find the system worthy to be kept alive and enhanced.

4.3.1 RUP Phases

Phases are terminated by milestones

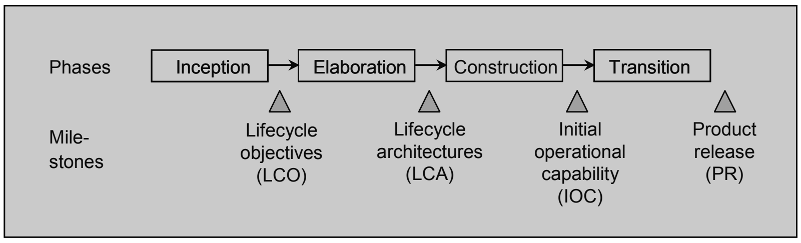
The phases divide a project into four major sections that are performed sequentially. Each section is terminated by a *milestone* supporting management controlling. Figure 4-14 shows the major milestones of RUP: life cycle objectives (LCO), life cycle architecture (LCA), initial operational capability (IOC) and product release (PR).

Go/no-go decisions

Milestones are points for go/no-go decisions. The stakeholders assess what has been and what can be achieved, and they decide how to proceed forward.

The four phases are briefly described below, following Rational Software's white paper on RUP [Rational 1998, pp. 3-7] and Ambler's summary of phases [Ambler 2005a, pp. 3-5].

Figure 4-14 RUP milestones


Phases
Inception
Elaboration
Construction
Transition
Mile-stones
Lifecycle objectives (LCO)
Lifecycle architectures (LCA)
Initial operational capability (IOC)
Product release (PR)

Inception phase

The goal of the *inception* phase is to achieve a stakeholder consensus regarding the objectives of the project and to obtain funding. The business case for the system is established and the project scope is delimited. The business case includes success criteria, risk assessment, estimates of the resources needed and a phase plan showing dates of major milestones. For this purpose, a high-level requirements model and perhaps a user-interface prototype are built.

Outcome of the inception phase

The outcome of the inception phase includes: a "vision document", i.e. a general vision of the core requirements, key features and main constraints; an initial use-case model; an initial business case, which includes business context, success criteria and financial forecast; an initial risk assessment; a project plan, showing phases and iterations; and one or several prototypes.

LCO milestone

The *life cycle objectives (LCO) milestone* at the end of the inception phase is the point where the stakeholders must agree on:

– the scope of the project,

– identification of the initial requirements, without much detail at this point,
– credibility of the cost/schedule estimates, priorities and risks,
– credibility that risk management is properly handled,
– the project plan is realistic,
– the software development process is appropriately tailored,
– the architectural prototype is pertinent.

A go/no-go decision is made based on the evaluation of these points. The project may be cancelled or considerably re-thought if it fails to pass the milestone.

Elaboration phase

The major objectives of the *elaboration* phase are to analyze the problem domain, establish a sound architectural foundation, develop the project plan and eliminate the highest risk elements of the project.

Requirements and architecture

Requirements and architecture are two important issues in this phase. Requirements must be specified in greater detail, although they will continue to evolve later. Decisions about the architecture have to be made with an understanding of the whole system: its scope, major functional and non-functional requirements such as performance requirements. They are based on an executable *architectural prototype* that at least addresses the critical use cases identified in the inception phase, which typically expose the major technical risks of the project.

The elaboration phase activities have to ensure that the architecture, requirements and plans are stable enough, and that the risks are sufficiently mitigated, so that it is possible to predictably determine the cost and schedule for the complete development process.

Outcome of elaboration phase: a use-case model

The major outcome of the elaboration phase is a comprehensive *use-case model* with descriptions of most use cases, supplemented by the non-functional requirements and any requirements that are not associated with a specific use case – or in other words, a software requirements specification. In addition, an operational description of the software architecture and an executable architectural prototype are created. The risk list, the business case and the development plan for the overall project, in revised versions, is a further outcome.

LCA milestone

The elaboration phase ends with the *life cycle architecture (LCA) milestone.* This is the point where the stakeholders must agree on:

– a realistic project vision and chance to succeed,
– a stable architecture,

- the requirements for the project,
- risks being properly managed and under control,
- current expenditures being acceptable,
- reasonable estimates for future costs,
- a sufficiently detailed iteration plan for the construction phase and
- the up-to-date overall project plan.

Again a decision to continue or cancel the project is required. The project may be aborted or considerably re-thought if it fails to pass this milestone.

Construction phase

Software system is made ready for deployment

During the *construction* phase, the system is developed to the point where it can be deployed. All remaining components and application functionality are developed, integrated and thoroughly tested. Requirements have to be prioritized, completely specified and analyzed. Based on the analysis, the solution is designed in detail, coded and tested. User feedback is obtained and taken into consideration for the final solution.

In large projects, parallel construction increments can be initiated to accelerate the availability of deployable releases. However, such increments increase the complexity of resource management and work synchronization.

The outcome of the construction phase is a software system ready for deployment with the necessary documentation (user manual, description of current release).

Initial operational capability (IOC) milestone

The third major milestone in the project is the *initial operational capability (IOC) milestone* at the end of construction. The stakeholders decide if the software, the installation and user sites, and the users are ready to go operational, without exposing the project to high risks. They must agree that:

- software and documentation are acceptable to be deployed,
- stakeholders are ready for the system to be deployed,
- risks are managed and under control,
- current expenditures are acceptable,
- estimates for future costs are realistic,
- the iteration plan for the transition phase is acceptable, and
- the up-to-date overall project plan is realistic.

Another go/no-go decision is due at this point. Although it is unlikely (yet not impossible) that the project will be entirely cancelled, transition may have to be postponed by another release if the project fails to reach this milestone.

Transition phase

Delivering the system to production

The focus of the *transition* phase is on delivering the system to production. This step requires first testing by both system testers and end-users, and any corresponding reworking and fine tuning. Operational databases have to be converted or connected to the new system. Before the software can be placed into the hands of the users (end-users, support and operations staff), these persons have to be trained in the new system. Typically, transition includes several iterations with beta releases, general availability releases as well as bug-fix and enhancement releases. The outcome of this phase is a running and working information system ready for productive use.

Product release (PR) milestone

At the end of the transition phase is the *product release (PR) milestone* where the stakeholders assess the state of the project. They must agree that:

– the software system, including supporting documentation and training, is ready for production; this includes the requirement that the system can be operated and that it can be supported appropriately once it is in production,
– current expenditures are acceptable,
– estimates for future costs are realistic, and
– the final system is complete and consistent with the project's vision.

Features of evolutionary development and prototyping

The RUP life cycle incorporates many characteristics of *evolutionary development* and *prototyping* that were proposed when these alternatives to a conventional waterfall model came into existence. Example of such characteristics are [Ambler 2005a, p. 5]:

– Work products – models, plans, source code, documents – evolve throughout the life cycle. Work products are not finished until the system is released into production.
– The project is planned in a rolling wave. This means that planning is detailed for immediate issues and less detailed for future issues. As the project progresses and tasks get closer, detailed planning for these tasks is done.
– Each phase ends with a go/no-go decision. Only if the stakeholders agree to move forward into the next phase is the project continued.

This may entail reworking the strategy for running the project. A project may be cancelled due to various reasons including quality concerns, lack of appropriate documentation, high deployment and/or support costs, or a shift in the strategic direction for the company.

– Risk management is built into RUP. Risks are documented and managed explicitly throughout the life cycle.

4.3.2 RUP Disciplines

An iteration addresses a portion of the entire system

Activities within the phases take place in so-called *iterations*. An iteration is a part of the project that addresses a portion of the entire system being developed. The result of an iteration, especially in the construction stage, is an *increment* or a subsystem which could conceivably be deployed to users as a release.

Actual project work takes place in "disciplines"

Within an iteration, the so-called *disciplines* are performed. (Disciplines were called workflows before. Neither of the two terms is truly appropriate. Since the actual project work is done here, "activity" or "task" might have been a better term.) In principle, all nine disciplines are relevant for each of the four phases. A typical view of an iteration is that a portion of the requirements is selected, analyzed, designed, coded, tested and integrated with the subsystem from earlier iterations.

The following description of the RUP disciplines is based on [Rational 1998, pp. 10-14] and [Ambler 2005a, pp. 6-9].

Business modeling

Goal: Creating a common understanding of the problem

The goal of *business modeling* is to create a common understanding of the business (or that part of the business that is relevant to the system being developed). Otherwise a frequently observed problem may occur: Business people and software developers do not speak the same language. They perceive problems from their respective points of view, describe them differently, and have their particular expectations how the problem should be solved. As a consequence, output from business engineering is not used properly as input to software development and vice versa.

Business modeling involves both parties

In order to avoid this shortcoming, business modeling as a RUP discipline involves both parties. The business people and the software developers document the relevant business processes with the help of business uses cases ensuring a common understanding among the stakeholders. Potentially, they can also identify ways for reengineering business processes. The result of business modeling is a domain model reflecting the relevant subset of the business.

Business modeling is a very important task for business information systems development, yet not necessarily relevant for other types of software development.

Requirements

Use cases and actors

The goal of the *requirements* discipline is to define what the system should do, i.e. the scope of the system. To achieve this, the desired functionality and constraints are elicited, evaluated and documented. *Use cases* are the primary means of describing the requirements found. Use cases represent the behavior of the system. *Actors* are identified, representing the users and any other system that may interact with the system being developed. The use-case description shows how the system interacts with the actors and what the system is supposed to do. Non-functional requirements are described in *supplementary specifications*.

In the iterations and phases of RUP, requirements continue to evolve. As new or changed requirements are identified, they may have to be prioritized. The use-case model developed in the requirements discipline is also used in other disciplines, e.g. during analysis and design, and test.

Analysis and design

Analyzing the identified requirements

In this step, the requirements are *analyzed* and a solution is *designed*. The meaning of "analysis" in RUP is somewhat different from other process models. Normally this term is used in the sense of "analyzing the problem in order to derive requirements". In RUP it means "analyzing the identified requirements" that were defined in the preceding discipline. A thorough understanding of the requirements is very important for the design of the new system.

Outcome is a design model

The outcome of analysis and design is a *design model*. This is a "blueprint" of how the source code is structured. It is based on an *architecture* which is elaborated and validated in the iterations. The design model exhibits components, subsystems, packages and classes. It also contains descriptions of how the objects will collaborate within the use cases. Design comprises not only system functions but also network, user interface and database design.

Implementation

Transforming the design model into executable code

The goal of *implementation* is to transform the design model into executable code. Source code in a programming language is written (or generated) for the classes and objects of the design model. Since writing code units and testing this code go hand in hand, *unit testing* is part of the implementation discipline. Code developed by different persons or teams has to be integrated into an executable system before a system test (see next discipline) can be performed. Integration is also done during implementation, involving more testing.

Test

In the *testing* discipline, proper working of the information system is investigated with the aim of achieving a high system quality. This includes finding and fixing errors in the programs, validating that the system works as designed, and verifying that the requirements are satisfied.

Planning test cases and strategies

System testing has to be planned, organized and documented. Test cases have to be specified and implemented. Strategies for automating tests may be defined. Testing is a comprehensive effort in all IS development projects. It will be discussed in more detail in section 6.3. RUP supports an iterative approach, which means that testing is done throughout the project. The goal is to find defects as early as possible, which significantly reduces the cost of fixing the defect. Testing supports primarily three quality dimensions: reliability, functionality and performance.

Deployment

Delivering the system and making it available to the end-users

The goal of *deployment* is to plan for the delivery of the system and to make the system available to the end-users. Since delivery normally refers to product releases, such releases have to be produced and shipped in the deployment discipline. Deployment covers a wide range of activities including the production of external releases; packaging, distributing and installing the software; and providing help and support to users. Deployment may also include activities such as planning and conducting beta tests, migration of existing data and formal acceptance by the customer.

Obviously the deployment activities are mostly taking place in the transition phase, yet many of the activities require preparation in preceding disciplines. Thus the deployment discipline is also spread across several phases, as the other disciplines are.

Configuration and change management (CM)

In any software development process, a large number of work products ("artifacts") are created. Use cases, design models and code modules are some of them. During the RUP iterations, different versions of these work products come into existence. One version of a module could be included in a customer release, while another is in test, and the third one is still in development. If problems are found in any one of the versions, fixes need to be propagated between them.

Critical activities of the *configuration and change management (CM)* discipline are centered around tracking versions and releases and managing and controlling changes in order to avoid confusion that leads to costly fixes and rework.

How to manage parallel activities, change requests, defect reports etc.

An important task of the CM discipline is to define how to manage parallel development, development done at multiple sites, and how to keep an audit trail on why, when and by whom any artifact was changed. CM also covers change request management, including how to report defects, manage them through their lifecycle, and how to use defect data to track progress and trends. It is very difficult, sometimes even impossible, to manually keep track of the large number of changes and requests. Therefore automated tools play an important role for configuration and change management.

Project management

Project management is integrated in RUP

Since *project management* and the software development process are closely connected, project management is integrated in RUP as one of its disciplines. Project management has to balance competing objectives, manage risk and overcome constraints to successfully deliver a product which meets the needs of both customers and users. Activities include scheduling, estimating effort, assigning tasks to people, monitoring work processes and results, tracking progress, controlling the budget, coordinating with stakeholders and many more.

The fact that many practical projects fail or are challenged is an indicator of the difficulty of the project management discipline. Project management is covered in detail in chapter 8.

Environment

Software development environment

The purpose of the *environment* discipline is to provide the project team with an appropriate software development environment – including both processes and software tools.

Customizing the unified process

In many cases the standard process (unified process) needs to be tailored to the needs of the organization or the project. RUP provides guidelines on how to customize the unified process to fit the specific needs of the adopting organization or project. Tools for tailoring the process are available, e.g. RMC (Rational method composer) [Kroll 2005] and EPF (Eclipse process framework) [Eclipse 2006].

Software tools include CASE tools such as an IDE or a collection of matching tools which support the major activities of the RUP disciplines, e.g. modeling, code generation, documentation, collaboration, configuration and change management. A suitable software environment has to be selected and installed for the project team.

Iterations

Disciplines are not strictly sequential

Disciplines as the steps of the iterations are not performed in a strictly sequential manner. In fact, activities often overlap. While the "natural" sequence: requirements → analysis → design → code → test etc. still exists, one activity does not need to be finalized before the next one can start. A more common approach is to take a subset of the requirements, do some analysis, go back to rework some of the requirements, proceed to analysis and design, rework requirements again, start coding, go back to design, etc. [Ambler 2005a, pp. 10-11].

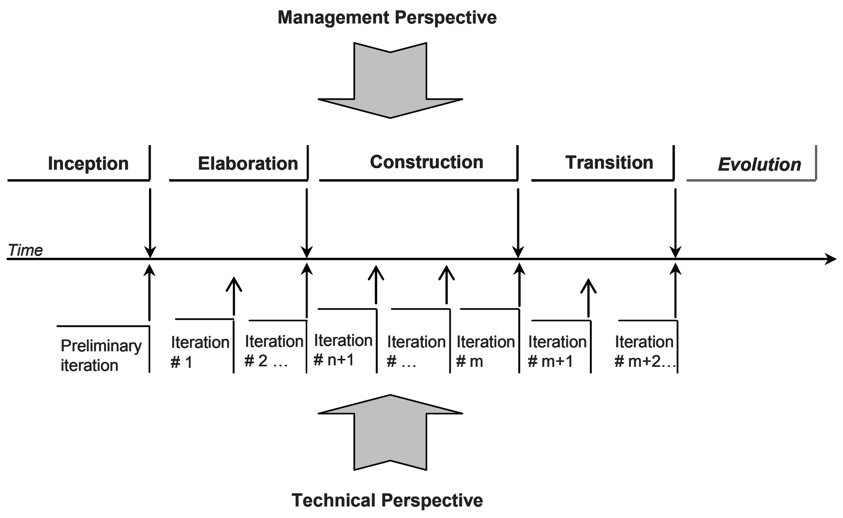
Iterations should be planned according to risk. This means that higher priority risks are addressed in earlier iterations and lower priority risks are addressed later. Likewise, immediate iterations are planned in greater detail while iterations which are parts of later phases are planned rather coarsely.

RUP perspectives

Management perspective vs. technical perspective

Disciplines in the iterations of RUP reflect the technical perspective of the process whereas phases approach the same process from a management perspective. The *technical* perspective deals with software engineering, quality and methodology aspects. The *management* perspective focuses on the commercial, financial, strategic and human aspects [Kruchten 1996, pp. 11-12]. Phases allow for management control and action while iterations allow for evaluation of technical artifacts, e.g. system releases and executable products.

Figure 4-15 RUP perspectives [Kruchten 1996, p. 12]



The management perspective and the technical perspective are not isolated but are aligned at the end of the phases. Since iterations are embedded in phases, a phase is over when the last iteration of the phase is completed. Figure 4-15 illustrates this way of synchronizing the management and the technical perspectives of RUP.

4.3.3 Best Practices

Successful approaches

The Rational unified process was derived from successfully practiced approaches to software development that have been used by many organizations. The creators of RUP called these approaches "best practices". RUP puts the best practices into an operational framework structured by phases and iterations. Best practices have been parts of RUP from the beginning on. With the need to adapt them to changing circumstances, the set of best practices was redefined in 2005 (see below).

Developing software iteratively

A key best practice proposed by Rational has always been to *develop software iteratively*. The underlying rationale for this best practice is that for today's sophisticated software systems it is not possible to sequentially first define the entire problem, design the entire solution, build the software and then test the product at the end. An iterative approach instead allows an increasing understanding of the problem through successive refinements, and an effective solution to grow incrementally over multiple iterations [Rational 1998, p. 2].

More best practices

Managing requirements throughout the process implies a systematic approach to eliciting, organizing, communicating and managing the changing requirements. The goal of this best practice is to reduce costs and delays [Kruchten 2001]. Use cases are the major instrument to describe requirements.

Using component-based architectures is a RUP best practice that accommodates change and promotes effective reuse, including a methodical, systematic way to design, develop and validate the architecture.

Visually modeling software focuses on the use of graphical tools and visual abstractions ("graphical building blocks" [Rational 1998, p. 2]). The visual language for this is UML.

Verifying software quality throughout the process refers both to product and process quality, building quality assessment into the process.

Finally, *controlling changes to software* is the ability to manage change, including making certain that each change is acceptable, and monitoring and keeping track of the changes.

As mentioned above, these six best practices have been the basis for RUP and thus for many organizations applying RUP. After a decade of experience and business-driven evolution, the six best practices have been re-thought and formally re-articulated in 2005 by the right-holders of RUP, IBM Rational (formerly Rational Software Corporation). The *updated best practices* are now:

Updated best practices (2005)

1) Adapt the process
2) Balance competing stakeholder priorities
3) Collaborate across teams
4) Demonstrate value iteratively
5) Elevate the level of abstraction
6) Focus continuously on quality

They are subsequently described, based on the official presentation by IBM Rational's management [Kroll 2005].

1) Adapt the process

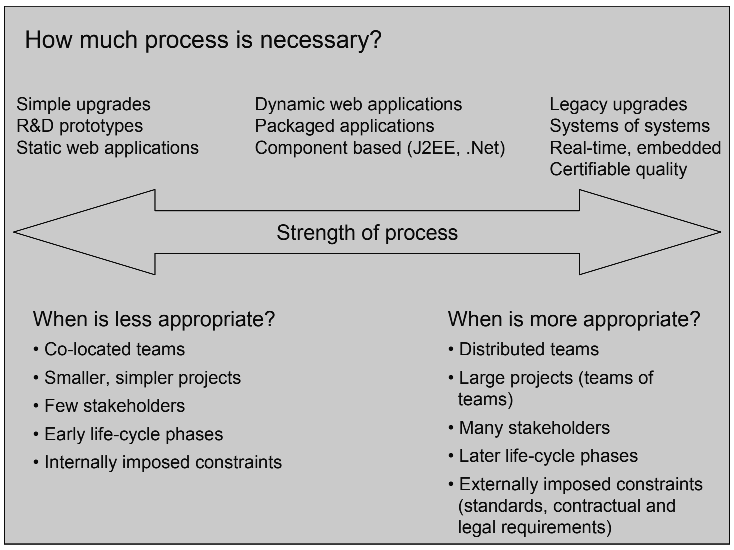
One process size does not fit all

Projects, organizations and people are different, so one process size does not fit all [Ambler 2005a, p. 13]. Instead the software development process needs to be tailored to meet the present situation. This best practice calls for adapting the process to the size and distribution of the project team, to the complexity of the application, and to the need for compliance. The latter may change during the development life cycle.

"More process leads to less creativity"

In the beginning of a project, uncertainty is typical, so creativity is desired and should be encouraged by a lean process. "More process typically leads to less creativity, not more, so you should use less process in the beginning of a project where uncertainty is an everyday factor." [Kroll 2005, p. 2]

Figure 4-16 Factors driving the amount of process discipline§


How much process is necessary?
Simple upgrades
R&D prototypes
Static web applications
Dynamic web applications
Packaged applications
Component based (J2EE, .Net)
Legacy upgrades
Systems of systems
Real-time, embedded
Certifiable quality
Strength of process
When is less appropriate?
• Co-located teams
• Smaller, simpler projects
• Few stakeholders
• Early life-cycle phases
• Internally imposed constraints
When is more appropriate?
• Distributed teams
• Large projects (teams of teams)
• Many stakeholders
• Later life-cycle phases
• Externally imposed constraints (standards, contractual and legal requirements)

§ Kroll 2005, p. 2.

In later iterations, however, more control may be preferred, such as change control boards, to remove undesired creativity and associated risk for late introduction of defects. The process will be more regulated and controlled.

Lightweight smaller projects

Likewise, for smaller projects with collocated teams and known technology, the process should be more lightweight. As a project grows in size, becomes more distributed, uses more complex technology, has more stakeholders, or needs to adhere to more stringent compliance standards, the process will become more disciplined. Project and software characteristics versus process discipline are plotted in figure 4-16.

Continuously improving the process

Adapting the process also means to *continuously improve the process*. An assessment at the end of each iteration and each project should be done to capture the lessons learned, and leverage that knowledge to improve the process. Finally, project plans and associated estimates should be balanced with the uncertainty of a project. This means that in the initial stages of a project when uncertainty typically is high, coarse-grained plans and associated estimates will suffice rather than plans and estimates of exaggerated precision. Early development activities should aim at driving out uncertainty to gradually enable increased precision in planning.

2) Balance competing stakeholder priorities

Balancing conflicting priorities is a difficult task

Different stakeholders – end-users, business management, operations staff, enterprise architects, external customers etc. – have different needs and priorities. Balancing these priorities, particularly when they often change in the course of the project, is a difficult task for the project manager.

As an example, most stakeholders would like to have a system that does exactly what they want it to do, while others insist on minimizing development cost and schedule time. These priorities are often conflicting. By using packaged software, for example, it is possible to deliver a solution faster and at a lower price, but only at the cost of trading off requirements. On the other hand, if an organization chooses to build the system from scratch, it may be able to address every single requirement, but both the budget and project completion date can be pushed beyond what is deemed acceptable.

Therefore it is important to understand the business processes and link them to projects and software capabilities. Based on this understanding, the business and stakeholder needs and subsequently the software requirements can be prioritized effectively. As the understanding of the system and the stakeholder needs evolve, the priorities may be modified in the course of the project.

Stakeholders' needs will evolve during the project

This best practice demands that development activities are centered around stakeholder needs. The project team must accept the fact that the stakeholders' needs will evolve during the project, just as the business is changing. In this process, the stakeholders develop a better understanding of the system's capabilities and which capabilities are the truly important ones to the business and the end-users. The development process needs to accommodate these changes.

Reusing assets

An important aspect of understanding how the stakeholders' needs can be satisfied is to understand what assets are available and to balance asset reuse with stakeholder needs. Such assets are, for example, legacy systems, services, reusable components and patterns. Reuse of assets can in many cases lead to reduced cost and higher quality.

Just as the above mentioned packaged software does, reusable assets may require a trade-off between costs and satisfying requirements. If reusing a component could lower development costs by 80 percent, but that component addresses only 75 percent of the requirements, the potential cost savings must be balanced with the stakeholder needs. Effective reuse may require such balancing in all phases of the project.

3) Collaborate across teams

The goal here is integrated collaboration across business, software-development and operation teams. As information systems become increasingly critical to running businesses today, close collaboration between those stakeholders deciding how to run the business, those developing the supporting information systems, and those running IT operations is indispensable.

In order to enhance team productivity and quality of results, people must be motivated to communicate and collaborate closely, and to actively learn new skills from their co-workers and other sources. Complex systems require the activities of different stakeholders with varying skills – business people, analysts, architects, developers, testers and operations staff. These people must be willing and interested in collaborating with one another across functions.

"Making a team commit to what they should deliver"

Motivating individuals on the team to perform at their best is an important first step. It includes "... making a team commit to what they should deliver and then providing them with the authority to decide on all the issues directly influencing the result" [Kroll 2005, p. 4]. The motivation of the team members is strengthened when they know that they are truly responsible for the end result. Each member needs to understand the mission and vision of the project.

Collaborative work environments

Collaborative work environments are a supportive means of fostering collaboration, including both software tools and physical workplaces and locations. Examples of software tools are tools for sharing work products, shared project rooms as well as tools for information, configuration and change management. Having developers and business people work together in close proximity, or even in the same room(s), is proposed as a means to foster collaboration [Ambler 2005a, p. 14].

4) Demonstrate value iteratively

Stakeholders see early what they will get

It is a good idea to deliver working software to the stakeholders early in the project to demonstrate the value of the new information system and to enable early and continuous feedback. This is done by dividing the development activities into iterations, each containing some requirements, design, implementation and testing, thus producing a deliverable that is one step closer to the final solution. End-users and other stakeholders see early what they will get. In some cases, they can use the software directly and provide fast feedback on the system's value and usability. It is not unlikely that they will realize that some requirements were forgotten, not correctly implemented, or just not meant the way they were interpreted by the designers.

Development teams should embrace change

Obtaining feedback early enables the project manager and the stakeholders to adapt the project plan. Most IS today are too complex to allow perfect alignment of the requirements, design, implementation and test in the first round. Instead, an effective development methodology has to embrace the *inevitability of change* [Kroll 2005, p. 6]. While traditional developers tend to dislike changing requirements once they have been defined, the unified process calls for development teams that embrace change and manage the change in the process.

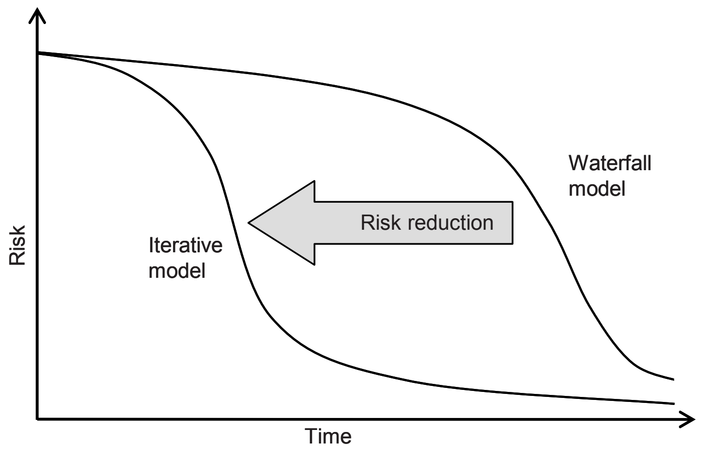
Seeking feedback to adjust plans

The underlying idea is that any requirements change, no matter how late in the life cycle, is welcome if it increases the information system's value, for example if it provides a competitive advantage to the company. It is beneficial to seek feedback early, and then adjust plans accordingly to ensure that the stakeholders will get what they actually need. Through early and continuous feedback, the development team learns how to improve the system, and the iterative approach provides the opportunity to implement those changes incrementally.

Continuously assessing risks

Underlying this best practice is also the need to drive out key risks early in the life cycle. This is illustrated in figure 4-17 comparing risk reduction in RUP and in the waterfall model. While in the latter one the majority of risks become obvious and can be treated only late in the process at a high cost, RUP makes it possible to address the major technical, business and programmatic risks early. This is done by continuously assessing what risks are still there, and addressing the top remaining risks in the next iteration. The major stakeholders should be involved in the risk assessment, in particular in the early iterations.

Figure 4-17 Risk reduction profiles [Kroll 2005, p. 6]



5) Elevate the level of abstraction

Working on a low abstraction level and using inefficient tools is a major obstacle to productivity. For example, writing all program code in a conventional programming language and creating a large database by hand-coding database structures directly in the DBMS's data definition language is cumbersome, time-consuming and error-prone. In fact, nontrivial IS today cannot be efficiently developed in this way.

Reducing the amount of human coding

Elevating the level of abstraction means to reduce the amount of human coding through higher-level models, tools and languages. This is the same goal as in model-driven information systems development discussed in section 4.2.3. Appropriate design and construction tools, for example, facilitate moving from high-level constructs to working code through automating or semi-automating design, construction and test tasks, and embedding integration and testing as seamless development activities in the process. In RUP, the models, tools and languages of preference are quite naturally those provided by UML (cf. chapter 5).

By using higher-level models and automated tools capable of generating lower-level models and/or code, developers can focus on more important issues like the system's architecture and quality. When the architecture is established early, a skeleton structure for the system is available, making it easier to manage complexity as more people, components, capabilities and code are added to the project.

Reusing existing assets

Reusing existing assets, such as reusable components, legacy systems, existing business processes, patterns or open-source software is also a means of raising the abstraction level. Reusable components, for example, can be considered black boxes to be used as they are. There is no need for the developers to deal with the component's internal complexity because the component is only accessed through its interface.

6) Focus continuously on quality

Quality is the responsibility of the entire team

Quality has a very high priority in RUP. Quality is the responsibility of the entire team, not just the testing team. Therefore, testing and validation is prevalent throughout the process. Every discipline includes reviews, either formal or informal, of the generated work products, and testing activities are critical to the implementation and test disciplines.

All team members contribute to quality

Team responsibility means that all team members contribute to enhancing the quality in all parts of the life cycle. *Analysts* have to make sure that requirements are testable. *Developers* need to make designs with testing in mind, and must be responsible for testing their code. *Managers* must ensure that the right test plans are in place, and that the right resources are in place for building the test environment and performing the tests. *Testers* are the quality experts. They guide the rest of the team in understanding software quality, and they are responsible for functional, system and performance-level testing.

In RUP the most important system capabilities are implemented early in the project. Towards the end of the project, the most essential software may have been in use and running for months, and it is likely to have been tested for months. Therefore an increase in quality is a primary tangible result of many RUP based projects [Kroll 2005, p. 8].

Building a testing environment incrementally

Building the *testing environment* goes hand in hand with building the system; i.e., the testing environment is also being developed incrementally. As the system is designed, consideration should be given to how it can be tested. Automating some (or finally all) of the testing activities is a goal pursued by the RUP community, and a key concern in agile development discussed in section 4.4.1.

The opposite of incremental testing throughout the iterations is to complete all unit testing before integration testing is done, which makes bug fixing and improving system quality a lot harder and costlier. This is typically the case when the process follows a waterfall-like model where testing is deferred to a separate phase late in the process.

4.4 Non-conventional Approaches to Information Systems Development

While the software life cycle and RUP are established process models, different approaches are also used today. These approaches can be considered as responses to changing goals (e.g. more freedom for developers) and ISD environments (e.g. availability of open-source software).

4.4.1 Agile Development and Extreme Programming

"A rebellion in the software industry"

"Recently there has been a rebellion in the industry against a growing tide of poor performances, long lead times, poor quality, disappointed customers, and frustrated developers. It is a rebellion against poor management. A passionate body of software developers has declared that there must be a better way – delivering software should be more predictable." [Anderson 2004, p. xxviii]

"Developers are disheartened by working ever longer hours to produce ever poorer software."

This quotation by David Anderson, an advocate of agile development (AD), expresses precisely the motivation for agile methods. After three decades of software engineering, the state of the art is still lamentable: Most development projects exceed budgets and delivery dates – provided that they don't fail completely. Maintenance costs are exploding due to low software quality, software developers are increasingly frustrated, and the reputation of their profession is continuously suffering. "Developers are disheartened by working ever longer hours to produce ever poorer software." [Martin 2003, p. 3] David Anderson articulates the consequences quite clearly: "Senior executives, perplexed by the spiraling costs of software development and depressed by poor results, poor quality, poor service and lack of transparency are simply shrugging their shoulders and saying, 'if the only way this can be done is badly, then let me do it badly at a fraction of the cost'. The result is a switch to offshore development and layoffs." [Anderson 2004, p. xxv]

"Agile alliance" – a counter-move

The above "rebellion" took place in February 2001 at a ski resort in the Wasatch Mountains of Utah, USA. A group of industry experts with long-standing software experience, naming themselves the "agile alliance," met there for skiing – and to somehow re-invent software engineering. Jim Highsmith, one of the participants, called them "organizational anarchists." [Highsmith 2001] The rebellion may be seen as a counter-move to the increasing industrialization of software engineering, and as a movement back towards the roots when software development was considered an "art" and software was hand-crafted.

"Manifesto for agile software development"

The most famous outcome of the meeting is the "manifesto for agile software development". This is a rather short position statement indicating what the agile alliance considers fundamental ideas for better software development. The manifesto is shown in figure 4-18.

Team work is valued

– Valuing *individuals and interactions* more than processes and tools puts a focus on the human factor. As software development is a team activity, collaboration is considered more important than a rigorously structured tool-supported process. Team work is valued higher than individual performance. "A team of average programmers who communicate well are more likely to succeed than a group of superstars who fail to interact as a team." [Martin 2003, p. 4]

"Produce no document unless its need is immediate and significant."

– The preference for *working software* over comprehensive documentation expresses an anti-position to thoroughly structured processes where documentation is overstressed and documents are given more importance than the actual target of the process, which is working software. Robert Martin even formulated this as his *first law of documentation:* "Produce no document unless its need is immediate and significant." [Martin 2003, p. 5] Short documents are preferred over long ones.

Collaboration with the customer

– *Collaboration with the customer* is considered more valuable than having waterproof contracts and communicating on legal terms. Customer feedback on a regular and frequent basis is indispensable for successful projects. The customer should be involved in the project, providing continuous feedback.

Responding to change

– *Responding to change* over following a plan is the same rationale as discussed for RUP. Change is inevitable and must be naturally incorporated into the process. The ability to respond to change appropriately may determine the success or failure of a project.

Figure 4-18 The agile manifesto [Agile 2001]

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

Software developers have a natural aversion against documenting

One comment regarding the dislike of documents may be added: Many documents are in fact just for the files, created because the process model or the project-management approach demands deliverables on paper. However, a natural aversion against documenting what has been done seems to be common characteristic of software developers. This aversion has been observed as long as software exists. Perhaps this attitude implicitly contributed to formulate the anti-documentation position.

Figure 4-19 The twelve principles of agile development [Agile 2001]

Principles behind the Agile Manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The agile manifesto is a concise summary of how software development in the authors' opinion should be approached. In a more operational view, the authors elaborated the rationale of the manifesto in twelve principles (cf. figure 4-19). Readers will recognize many good ideas that are present in other post-waterfall-model approaches as well.

Agile methodologies

While the agile manifesto and the agile principles express a certain way of thinking about and attacking software development, they provide neither an operational approach nor a process model. Several methodologies have been developed to fill this gap. Some actually existed before the agile alliance articulated the manifesto and influenced the formulation of the agile principles. Among the agile methodologies are:

» Extreme programming (XP)
» Scrum (http://www.controlchaos.com/about/)
» Feature driven development (FDD, http://www.featuredrivendevelopment.com/)
» Crystal Clear [Cockburn 2004]
» Adaptive software development (ADP, http://www.adaptivesd.com/)

Extreme programming (XP)

Kent Beck, Ward Cunningham and Ron Jeffries

Extreme programming is the best-known of the agile methodologies. It was developed by three participants of the agile-alliance meeting, Kent Beck, Ward Cunningham and Ron Jeffries, in the late 1990s, and became popular through Kent Beck's book in 1999 (first edition of [Beck 2004]).

XP is a set of simple and concrete practices

XP stresses customer satisfaction. The main goal of XP is to reduce the cost of change. In traditional system development methods, requirements are determined in the beginning and often fixed from that point on. As pointed out earlier, the cost of changing the requirements at a later stage can be very high. XP sets out to lower the cost of change by introducing basic values and principles, making the process more flexible with respect to changes.

Based on these values and principles, extreme programming is a set of simple and concrete practices that combines into an agile development process [Martin 2003, p. 17]. XP is based on the values and principles of:

– *Communication* – frequent and extensive, both within the project and with the customer.
– *Simplicity* – always starting with the simplest possible solution and turning it into a better one later.
– *Feedback* – frequent and rapid feedback from the system (by unit testing), from the customer (by frequent acceptance tests), and from the team (by involving the team into requirements changes immediately).
– *Courage* – for example, knowing when to create a better solution or when to throw code away.

– *Respect* – team members respect each other, avoiding changes that will delay the work of their colleagues. Members respect their work by always striving for high quality and the best solutions.

XP practices

In the following, the practices of extreme programming incorporating the above values and principles are briefly explained. This outline is based on XP descriptions where the interested reader will find more details [Beck 2004, Beck 2005, Wells 2006, Martin 2003].

Customer team member

➠ *Customer team member ("the customer is always available"):* All phases of an XP project require communication with the customer. Face-to-face communication onsite is preferred. One or more customer representatives should be on the development team.

User stories

➠ *User stories:* These are substitutes for large requirements documents, used to create quick time estimates for scheduling work and release planning. They are written by the customers and explain what they need the system to do for them. User stories are not meant to capture all details. Instead, they should only provide enough detail to make a reasonable time estimate for implementation. When the time comes to implement the story, developers will talk to the customer and receive a detailed description of the requirements face-to-face.

Short cycles, small releases

➠ *Short cycles, small releases:* XP projects should deliver working software in iterations, for example every two weeks. The team seeks to discover small units of functionality that make good business sense and can be released into the customer's environment early in the project. After a number of iterations, a release is planned. A release is a major delivery that the customer can put into production, comprising the work of about three months. A release plan consists of a prioritized collection of user stories that have been selected by the customer. With releases and iterations, the developers get valuable feedback early enough to have an impact on the project's progression. This is preferable because the longer a team waits to introduce an important feature to the users, the less time they will have to fix any problems.

Acceptance tests

➠ *Acceptance tests:* The details about user stories are captured in the form of acceptance tests, i.e. the user stories are translated into acceptance tests. The customer specifies scenarios to test when a user story has been correctly implemented. Acceptance tests are black-box tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and for deciding on which failed tests are of highest priority.

Pair programming

➠ *Pair programming:* All production code to be included in a release is created by two people working together at a single computer. One member of the pair types the code and uses the mouse, for example, while the other member watches the code being typed, looking for errors and improvements, and perhaps giving strategic thoughts to the implementation. Pair membership should change at least once per day. Eventually each team member will have worked with every other team member. Pair programming has been observed to increase software quality without impacting the time to deliver[§].

Test-driven development

➠ *Test-driven development:* Testing is planned and performed in a non-conventional way. Unit tests are one of the cornerstones of XP. A *unit test framework* like JUnit (http://www.junit.org) is useful to enable developers to create automated unit tests suites. The tests are created before the code is written ("test first" principle), so the goal of unit testing can be characterized as making failing unit tests pass. (Obviously a test done on non-existing code will fail, yet when the code is written, it will eventually make the test pass.) Test cases are usually written shortly before the code is written, so the body of test cases grows with the code. Unit-test frameworks are considered development tools in XP just as a compiler and an editor are. When a pair makes a change to a unit, they can run the associated tests again to ensure that nothing was broken. Binding development progress to units and isolated testing of units supports the decoupling of units and thus the fundamental idea of object-oriented development: encapsulation.

Collective ownership

➠ *Collective ownership:* The entire team is responsible for the entire system, its code, its architecture etc. A pair has the right to check any module and improve it, fixing bugs or refactoring. Combined with automated testing, anyone can make a change to any piece of code and release it to the code repository as needed. Before any code is released it must completely pass the entire test suite. No one person becomes a bottleneck for changes, and everyone may contribute new ideas to all segments of the project.

§ A side observation: In a pair programming experiment conducted with students of computer science, McDowell et al. found that those who paired got significantly higher scores for their programs than those who worked alone. The percentage of students who passed the final exam was higher in the pair-programming groups than in the non-pairing groups [McDowell 2006].

Continuous integration

➟ *Continuous integration:* Developers should integrate and release code into the code repository rather often, every few hours. Changes should never be withheld for more than a day. In XP, pairs are encouraged to check out any module any time, improve it and check it back in (collective ownership). Parallel pairs are likely to work on the same module. So the question is: How will parallel changes to the same module be handled? Or in other words, how will the changes be merged? The rule in XP is simple: sequential integration. Only one pair integrates at any given moment. The first pair to check in wins, everybody else has to merge their changes onto the last checked-in version. Continuous integration avoids diverging or fragmented development efforts. Everyone needs to work with the latest version so that changes are not being made to obsolete code. Frequent integration reduces the potential problems substantially.

Sustainable pace ("no overtime")

➟ *Sustainable pace ("no overtime"):* XP teams are not allowed to work overtime. "Projects that require overtime to be finished on time will be late no matter what you do." [Wells 2006] Instead, it is recommended to have a release planning meeting to change the project scope or timing when delivery dates are endangered. The only exception to the rule is the last week before a release if the team is very close to reaching its release goal. In this case overtime is permitted.

Open workspace

➟ *Open workspace:* To facilitate communications the team works in an open workspace with all the team members and equipment being easily accessible. People who sit in pairs in front of workstations can communicate intensely. Although counter-intuitive, it has been found that despite noise and distraction, such an environment may increase productivity by a factor of two.

Planning game

➟ *Planning game:* Since XP is an iterative development process, the goals of the next iterations must be planned. In the planning game, responsibilities are divided between the customer and the developers. Together they determine the scope of the next release. Customers select features (user stories) and decide how important these features are for them. Developers estimate how much the story will cost to implement in terms of person weeks and what is feasible in the next iteration or release. The customers then decide what story is the most important or has the highest priority to be completed.

Simple design

➟ *Simple design:* "A simple design always takes less time to finish than a complex one." [Wells 2006] The design is kept as simple as possible for the current set of stories. Only the stories for the current iteration are considered, not any future stories. Instead of creating an overall system design, the developers start with a simple design and migrate that design from iteration to iteration, to be the best design for the current set of stories. Thus an XP team will not start with building frameworks and infrastructure for the features that might be coming, but take the first set of stories and make them work in the simplest possible way. Developers strive to keep things as simple as possible for as long as possible by never adding functionality before it is scheduled.

Refactoring

➠ *Refactoring:* As programmers add new features to the project, the code tends to degrade and the design deteriorates. If this continues, the code will end up in an unmaintainable mess. So from time to time, refactoring is necessary. Refactoring is a process of incremental improvement. In a series of small transformations, the structure of the system is improved without affecting the system's behavior. After each small transformation, the unit tests are run to make sure that nothing was broken. In this way the system will continue to function while the design is transformed. Refactoring should be done continuously throughout the project rather than at the end of a release, an iteration, or a day. "Refactoring is something we do every hour or every half hour." [Martin 2003, p. 16]

Metaphor

➠ *Metaphor:* Choosing a system metaphor helps people to get the big picture of the system. The system metaphor provides an idea or a model for the system. In particular, it provides a context for naming things (e.g. classes, objects) in the software consistently. Consistent names are very important for understanding the overall design of the system and for code reuse as well.

Further recommendations

In addition to the practices explained above, XP advocates provide more rules and recommendations on how to work in a development project, including:

– Agreeing on coding standards to keep the code consistent and easy for the entire team to read and refactor.
– Moving people around and having them work on different sections of the system, to avoid knowledge loss and coding bottlenecks.
– Using CRC (class, responsibilities and collaboration) cards to design the system as a team; typically used to determine which classes are needed and how they will interact.
– Having a stand up meeting every morning to communicate problems, solutions, and promote team focus.

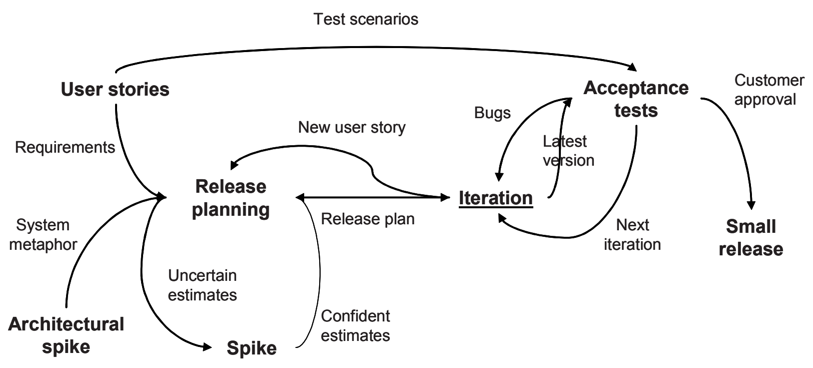
– Optimizing last ("make it work, make it right, then make it fast" [Wells 2006]) – no time should be spent optimizing the code until the end because it will change continuously.

XP process model

Agile developers dislike "process"

Does extreme programming have a process model? In the agile community, "process" is often used with a negative undertone, implying that a process has attributes such as "disciplined" or "structured" which are disliked by agile developers. These attributes are associated, for example, with the waterfall model and with iterative processes in cases where they are planned and structured. In contrast to this, XP is described in terms of values and practices.

Figure 4-20 XP process model [Wells 2006]



Nevertheless, the description of relationships between iterations, releases, stories and tests indicates that there is a certain process idea underlying extreme programming. Figure 4-20 illustrating the flow of work can be interpreted as a generic model of XP development processes. A concrete project is characterized by a particular path through the graph.

Spikes are simple prototypes

Spikes are simple prototypes – solutions created to figure out answers to tough technical or design problems. Most spikes are not good enough to keep, so they are usually thrown away. The goal is reducing the risk of a technical problem or increasing the reliability of an estimate for a user story.

Agile vs. structured?

Followers and opponents

Since its first articulation in 2001, agile development has found a large community of followers – and fierce opponents as well. While the followers are fascinated by the human-centric approach of agile development, opponents criticize that agile development is contrary to a disciplined process and impossible to control. (AD advocates reply that there is no need for control in motivated teams.)

Agile development and structured approaches seem to be so far apart that a veritable method war broke out. Grady Booch, in the foreword to Boehm and Turner's book on a possible comprise, expects "... that this won't be the last set of method wars I'll live through." [Boehm 2004, p. xiii] Barry Boehm, a software engineering pioneer, and Richard Turner, an author on the original SEI-CMMI team [SEI 2007], compared both agile and disciplined (sometimes called "plan-driven[§]) development in their book entitled "Balancing Agility and Discipline – A Guide for the Perplexed". At the end of their examination, the authors drew six conclusions [Boehm 2004, p. 148]:

"Balancing agility and discipline"

1. Neither agile nor plan-driven methods provide a silver bullet.
2. Agile and plan-driven methods have some home grounds where one clearly dominates the other.
3. Future trends are toward application developments that need both agility and discipline.
4. Some balanced methods are emerging.
5. It is better to build your method up than to tailor it down.
6. Methods are important, but potential silver bullets are more likely to be found in areas dealing with people, values, communication and expectations management.

§ "Plan-driven" means that the development process follows a plan with exactly defined stages, activities and documents accompanying the activities. There is a concern for completeness of documentation at every step so that thorough verification of the results is possible. Not only the waterfall model but also incremental and evolutionary process models fall into this category if they mandate strong documentation and traceability [Boehm 2004, pp. 10-11].

4.4.2 Reuse-oriented Process Models

Reducing the development effort

Reuse of existing software components is a concept underlying many development projects. The obvious goal is to reduce the development effort, thus leading to lower cost and shorter development time. Information systems development using a service-oriented architecture, for example, is explicitly based on the reuse of components, i.e. web services or enterprise services. In general, reusable components may be found from an organization's earlier projects, from open-source libraries, from UDDI registries on the Internet (see section 3.3.1) and from software vendors "off the shelf".

COTS oriented process model

Commercial-off-the-shelf components

The latter type of components is known as *COTS (commercial-off-the-shelf)* components. Diverse kinds of software have been summarized under this term. A common understanding is that a COTS component is a prebuilt piece of software supplied by a vendor, that is integrated into the software system under development [Morisio 2002, p. 189]. The component becomes a part of the new system. It must be there to provide operational functionality of the system.

Screening available components and gluing them together

Major tasks in a COTS based development effort include screening of available components as to their suitability against existing requirements, gluing components together, and building pieces of software that are not available as COTS [Conradi 2003].

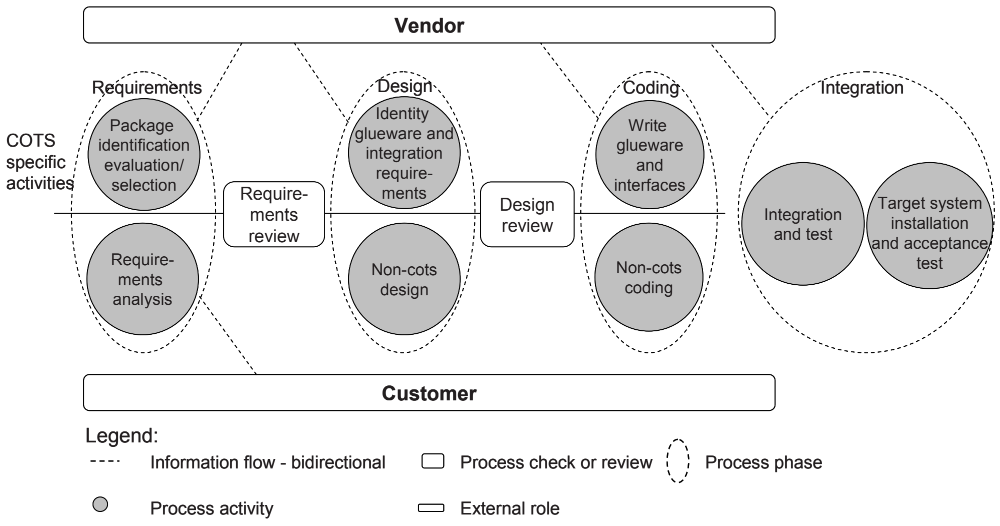
In the original COTS approach, components are thought of as being more or less "shrink wrapped". This means that a component must be taken as it is (no customization). If it only comes close to the requirements of an individual organization but does not meet them completely, either the requirements have to be adapted or some functionality has to be developed outside the COTS component.

Morisio et al. studied the actual processes of 15 NASA (National Aeronautics and Space Administration) projects using COTS for the development of satellite ground support software [Morisio 2000]. Based on this research, they proposed a process model for COTS based development in which several COTS components plus a considerable amount of new developed software have to be integrated.

Main phases of COTS based development

The main phases shown as dashed ovals in figure 4-21 are: requirements, design, coding and integration. Most phases encompass specific COTS based activities. These activities are drawn above the horizontal line in figure 4-21. Conventional activities are placed below the line. Major activities in the four phases are the following ones [Morisio 2002, pp. 195-197]:

Figure 4-21 Process model for COTS based development[§]



Conventional and COTS specific activities

– *Requirements:* The requirements phase now comprises both conventional analysis activities and COTS specific activities. The part specifying the activities of COTS identifies and evaluates available components using vendor documentation, reviews, peer experiences and other sources. A feasibility study including a complete requirements definition and an effort estimation may be conducted, and a high-level architecture and a risk assessment model may be developed. At the end of the phase, the initial requirements are reviewed

§ Morisio 2002, p. 195.

and perhaps adapted in light of what is feasible with the selected components.

Main concern is integration of COTS components

– *Design*: The design phase encompasses a high-level design where the main concern is defining the integration of COTS components and new developed software. This may be particularly demanding when several components are involved, each one with possibly different architectural styles and constraints. Requirements for so-called *glueware*, i.e. software to bind the components together, are specified. Glueware may be needed to invoke the components' functionality, to do exception handling and to resolve incompatibilities between two components. If it becomes clear in the design review at the end of the phase that integrating the selected components is impossible, the process goes back to the requirements phase (an arrow not explicitly shown in Morisio et al.'s process model).

Non-COTS modules, glueware and interfaces

– *Coding:* This phase covers primarily the coding of non-COTS modules, glueware and other interfaces between COTS components and conventional software. The overall amount of coding will obviously be significantly lower than in a traditional development project.

Actual integration of the COTS components

– *Integration:* Actual integration of the COTS components, with the help of the glueware developed, and of the non-COTS modules is the subject of the integration phase. While in theory this should be a fairly easy step as everything was decided and assessed beforehand, reports from practical projects indicate that integration consumed the most effort [Morisio 2002, p. 194].

Advantages and disadvantages

The *advantages* of COTS based development include lower costs, less work and shorter completion times. Building on existing components can also enhance the reusability of the new solution.

Obvious *disadvantages* are: 1) that the customer's requirements are likely to be "smoothened", i.e. adapted to what the components are capable of providing; 2) interfacing components with other components may be technically complicated; and 3) the debugging of the final system can prove to be very difficult because the COTS components are black boxes.

Reusing web services

While the original COTS approach was targeting conventional software technology and components based on, for example, MS Excel or Access, object-oriented technology and later web services have opened up new opportunities for reusing existing software.

In particular, UDDI registries and other sources on the Internet nowadays offer components in the form of web services or enterprise services. Reuse-oriented development employing a service-oriented architecture basically encompasses the same steps as shown in figure 4-21: Available services have to be found, evaluated, selected, composed and integrated with other software modules.

WSBPEL providing the glueware

The role of the glueware is played by a language that lets the developers define how services have to be invoked in the execution of a business process. *WSBPEL (web services business process execution language)* is an example of such a language used to specify business process behavior based on web services [OASIS 2006]. The activity of composing web services for larger services and finally complete solutions for an entire or a partial business process is called *web services orchestration (WSO)* [Newcomer 2004, ch. 6].

4.4.3 Open-source Software Processes

Does open-source software (OSS) development follow a process model? Although OSS development is not the topic of this book, some open-source software characteristics should be noted. To discuss the above question, we take up Fitzgerald's distinction between FOSS and OSS 2.0:

– FOSS ("free and open-source software") – software created by many volunteers (or what has been called a "crowd of anarchist programmers") collaborating on the Internet, and
– OSS 2.0 – software created by professional organizations in a more rigorous way and published in source code [Fitzgerald 2006, p. 587].

FOSS development

Based on ideas by Eric Raymond

FOSS development has been based on rather idealistic ideas formulated and revised from 1997 on by Eric S. Raymond. Stimulated by an investigation into how the Linux operating system came into existence, Raymond turned his insights into 19 principles, some of which are noted below [Raymond 2000]:

1) Every good work of software starts by scratching a developer's personal itch.
2) Good programmers know what to write. Great ones know what to rewrite (and reuse).
3) "Plan to throw one away"; you will, anyhow.§
6) Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.
7) Release early. Release often. And listen to your customers.
8) Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. "Given enough eyeballs, all bugs are shallow."#
19) Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

Involving many people increases the software quality

While the second and third principles reflect established software-engineering knowledge regarding system evolution, principles 6, 7, 8 and 19 indicate the general direction of FOSS development: involve many people in the development, as it increases software quality because bugs will be found and fixed quickly.

Two general characteristics of FOSS development processes are: 1) a strong focus on incrementally writing, testing and debugging code, and 2) less focus on the early process stages (planning, analysis, design). The decision to make new software is not a management act based on a project proposal, but a developer's "itch worth scratching" (see principle no. 1). Often, the same person performs the analysis, designs and writes the first prototype.

"No need for a lengthy discussion of requirements"

This is in sharp contrast to conventional software engineering approaches where the most importance is placed upon the analysis and design stages, while implementation and testing are considered subordinate activities. FOSS protagonists like Eric Raymond see no need for a lengthy discussion of requirements since requirements are taken as understood – an assumption that is true if developers and end-users are the same.

Starting point for the community to come in

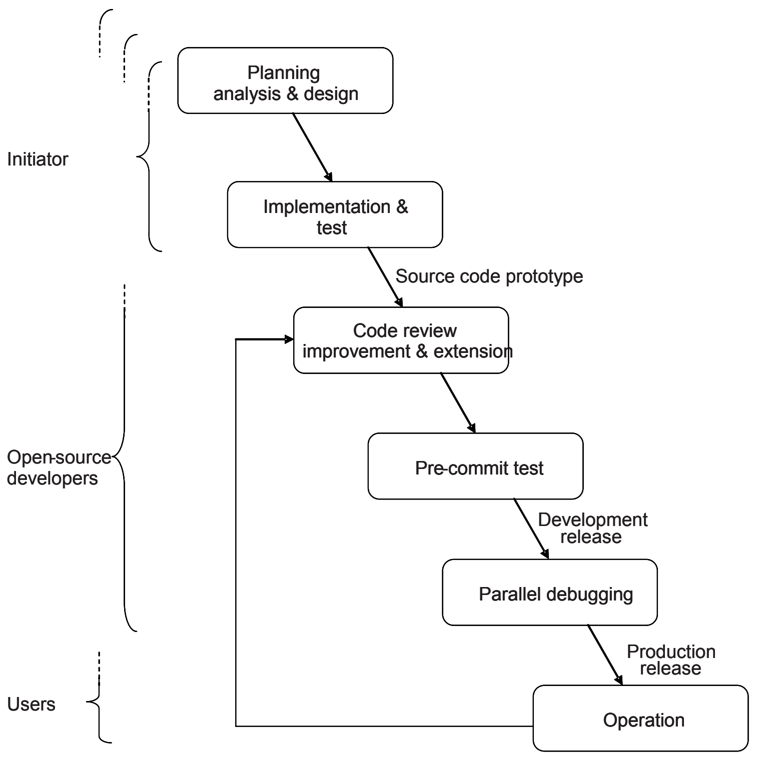
The first prototype published on the Internet is actually the starting point for the OSS community to come in. As shown in figure 4-22, existing code is reviewed by interested volunteers and improved, and new code is incrementally added. Before a new release is published on

§ The quote is the title of chapter 11 of Frederick Brooks's book "The Mythical Man-Month [Brooks 1995, p. 115].

# Raymond called this quote by Linus Torvalds, the developer of Linux: "Linus's Law" [Raymond 2000, p. 8].

the Internet for review and further development, pre-commit tests are encouraged. The development release is tested and debugged in parallel by a potentially large number of developers worldwide, making bug finding and fixing a short period. A stable, debugged production version can be released subsequently [Fitzgerald 2006, pp. 588-589].

Figure 4-22 FOSS development process model



Note that in contrast to conventional software development, the initiator can also be an open-source developer. Likewise developers are often users of the system at the same time. This is indicated by extended braces in the figure.

Developers and users are often the same

"Bazaar-style" approach vs. "cathedral-style" approach

The assumption underlying the FOSS approach is that motivated, powerful individuals make idealistic contributions in the development of a software system. This assumption has been discussed and questioned by many authors. Nikolai Bezroukov, in an often cited essay on "Open Source Software Development as a Special Type of Academic Research", seriously challenged most of Raymond's assertions [Bezroukov 1999], initiating an academic discussion on how OSS development should and actually is being done. The discussion has been going on since then. In fact, rather than developing software following Raymond's pure "bazaar-style" approach [Raymond 2000], a more centrally coordinated way is often preferred ("cathedral-style.")

OSS 2.0 development

A paradigm shift in the OSS market

Since the beginning of Linux, the development of open-source software has undergone a fundamental change. In particular, business firms and commercial organizations have entered the open-source community, making formerly proprietary software open-source or sponsoring the development of new open-source software. This has brought a radical paradigm shift in large sectors of the OSS market.

The ultimate goal of the new players is obviously not to give things away for free but to earn money. One way of creating revenue is to offer support and services around a software product which is as such license-free.

Planning OSS has become a strategic business activity

Another motive for supporting open-source software is due to its strategic potential to alter the competitive forces at play. "The haphazard principle of individual developers perceiving 'an itch worth scratching' is superseded by corporate firms considering how best to gain competitive advantage from open source. ... For example, IBM is a strong supporter of Linux, because it erodes the profitability of the operating system market and adversely affects competitors like Sun and Microsoft" [Fitzgerald 2006, p. 591]. In this light, planning an OSS product is a strategic activity.

Capturing requirements is an important phase

When developers and end-users are the same, as assumed in the FOSS community, requirements and the system design may, in fact, be taken as understood. However, when business information systems or other business software are the subject of the development effort, then an organization's employees and not the developers are the end-users. Consequently, capturing the requirements and transforming them into meaningful system functionality based on an appropriate architecture are much more deliberate phases in OSS 2.0 than in FOSS.

Formalized meetings and conferences

As a consequence, management of the development process is less bazaar-like. In order to achieve a professional product, *analysis* and *design* need much attention. For a number of widely used open-source products, formalized meetings have been established, for example the Apache Foundation's conferences in the United States and in Europe (http://www.apache.org/foundation/conferences.html). These meetings bring together developers to coordinate and plan further development of the respective product [Fitzgerald 2006, p. 591]. As a consequence, extensions and new features of the system are not created in a pure bazaar-like style as in FOSS development but in a more coordinated way.

Paid developers working on open-source products

The basic cycle of figure 4-22 is still being employed in OSS 2.0 development. Researchers investigating open-source software note, however, that the bazaar-like style – many people working on the same product – is being less applied to the development process and more to the product-making process (product stabilization, delivery, support) [Fitzgerald 2006, p. 593]. This shift is understandable when taking into account that nowadays large stakeholders are assigning paid developers to work on open-source products.

Kim Johnson, whose master's thesis is probably the most cited reference on OSS process models [Johnson 2001], names as an example Microsoft's strategy of shipping early versions of products that are notoriously bug ridden:

"As long as a product can demonstrate plausible promise, either by setting a standard or uniquely satisfying a potential need, it is not necessary for early versions to be particularly strong" [Johnson 2007]; or in other words: the Microsoft community will do a good share of the testing, helping Microsoft in the debugging and making of a stable product.

4.5 Offshoring Process Models

Two organizations are building the IS

Offshoring is different from the previously discussed approaches and process models insofar as not just *one* organization but *two* are building the information system. A subsidiary established by a mother company (captive center) is also considered a second organization. Operating in a different social, cultural and legal environment, a captive center is subject to different organizational factors than the mother company.

Underlying the above process models was the implicit assumption that either: 1) the user organization itself is developing the system, 2) a software company commissioned for the job is developing the system, or 3) a software vendor is producing standard software for the open market. A domestic software company building a custom information system for another organization is, of course, also a second organization but in this case the contractor is likely to work in a similar way as an internal ISD group, being more or less in daily contact with the customer.

Communication and interaction must be planned thoroughly

In offshoring projects, the organization carrying most of the development work is at a remote location, far away (offshoring) or at least not close (nearshoring) so that daily face-to-face communication is not a typical characteristic. This means that communication and interaction need to be planned and ensured in a different way. Transferring work from one organization to another one requires the work to be based on well-defined documents, software pieces and quality assurance.

Process models for offshore and onshore ISD are similar

At first sight, process models followed by organizations that offshore information systems development are not much different from models for onsite development. Most offshoring process models in practice are based on one of the previously discussed models, yet with specific extensions to capture offshoring needs and reality. In particular, organizations that developed software themselves before they started offshoring often continue to use the same process model as before. An obvious reason is that available project-management experience and knowhow are related with that model.

The offshore company's model must match the offshorer's model

Offshoring providers, on the other hand, are somewhat limited in the choice of process model for their part, because their model has to match the offshorer's model. The looser the connections between the client and the offshore organization are, the more freedom the latter one has to proceed according to its own needs and preferences. Obviously the offshoring provider cannot follow an iterative approach like RUP if the client's process model is strictly sequential.

4.5.1 The Offshorer's Perspective

Typical offshorers

Organizations employing an offshore software company or a captive center for information systems development are typically either:

- user organizations with their own development groups, attempting to reduce cost,
- software organizations developing custom information systems, with the user organization involved in the development,
- software organizations developing standard software.

Atypical offshorers

Rather atypical offshorers would be user organizations that neither have their own development group nor a commissioned software firm for the development effort. Although not inconceivable, organizations without software development knowhow are unlikely to turn to remote offshoring providers directly. They are more likely to look for help from a domestic firm (who in turn may send some of the development offshore).

Sequential process models are dominating

Since many organizations still apply a *sequential process model* or a variant of this model in their development projects, it is not surprising that sequential approaches are dominating the published offshoring process models. Notwithstanding its many disadvantages, the waterfall model provides clear milestones, deliverables and points of management control that are particularly helpful when communication and feedback are by nature not as close as in an onsite project.

High-level offshoring process models often exhibit the same sequential phases as a conventional process model for onsite development, for example:

- Business problem definition
- Requirements
- Analysis and design
- Implementation
- Testing
- Deployment
- Operations and support

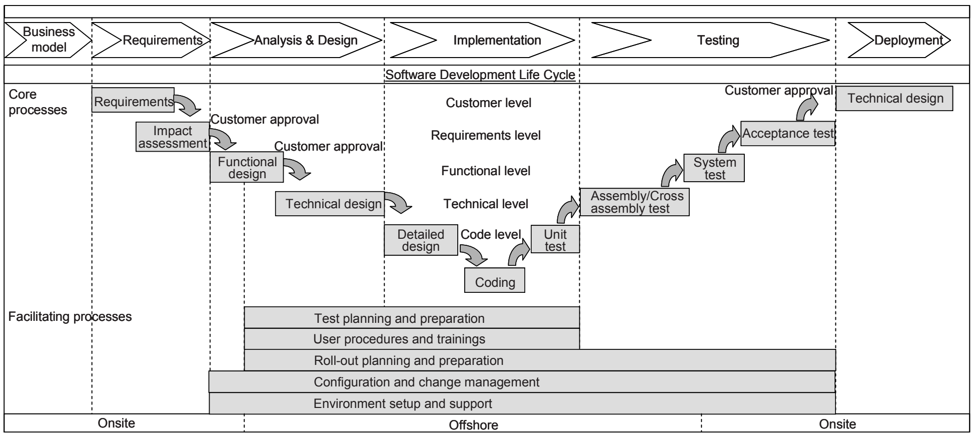
Differences lie in the responsibilities for each phase's activities (who does what?), including additional activities not present in conventional process models. Responsibilities for activities depend on the chosen scope of outsourcing as discussed in section 2.3.5. This means, does the organization wish to send offshore:

1. coding and testing only,
2. module design, coding and testing,
3. system design, module design, coding and testing,
4. "the problem"?

Responsibilities depend on the scope of outsourcing

In the first and second cases, most life cycle tasks remain with the customer's project team. In the third case, the cut-off stage is the system design. Often the onsite and offshore teams collaborate for the design as illustrated in figure 4-23.

Figure 4-23 Process model with onsite and offshore responsibilities



This figure reflects the high-level process model of a large, globally acting software company. The underlying division of labor leaves project management, architecture, high-level design, risk and quality management, and the final testing responsibilities with the offshorer while the detailed design, coding and much of the testing are the responsibility of the offshore-services provider.

The fourth case, outsourcing the entire business problem, is again different in that most of the responsibilities are with the offshore organization. This organization is likely to employ a different business model than in cases 1, 2 and 3. In section 4.5.2 we will look into the fourth case more closely.

Examining offshoring process models in more detail exhibits a number of additional tasks that are different from conventional process models. They are largely due to additional communication, collaboration and control requirements between onsite and offshore personnel which are not present in onsite projects.

Is the project suitable for offshoring?

One such task is examining whether the project is suitable for offshoring at all. Figure 4-24 lists some of the offshoring criteria. One such criterion is management commitment: Is the offshorer's management committed to send the development offshore? If the offshorer is a software company developing the system for a customer, the customer might object to offshoring. Small projects are usually not suited to offshoring because of the additional overhead as compared to onshore development.

If the system cannot be divided into separate work units, then it is difficult to send portions of the project offshore and reliably control deliverables. Usage of standard technologies makes offshoring easier than usage of proprietary technologies. A system with tight connections and many interfaces with other systems in the customer's IS environment raises more difficulties than a system that is only loosely coupled. If the offshore provider needs to know and understand the customer's business process in detail, it might be too hard to transfer all that knowledge from the customer to the provider.

The role of documents

Communication between the onsite and offshore teams is inevitably and to a significant extent based on documents. Therefore the availability and quality of relevant documents play an important role. The better the requirements and design specifications are, for example, the less misinterpretation will occur on the developers' side. The languages on both sides may pose a problem for communication if the language is not the same and the language barrier cannot be overcome.

Additional communication due to approval sequence

An example of additional communication requirements is a sequence of approval steps as illustrated in figure 4-25. The process summarized in this figure is practiced by DCandM, a Brazilian offshoring provider [DCM 2006]. Here it is assumed that the offshore company enters the process at a very early stage, when the business problem is investigated and business requirements are being specified. The specification created by the offshoring provider's consultant specifies the functional requirements from a business perspective.

This specification is also created in several steps and iterations, with onsite and offshore technical staff involved. Finally it has to be approved by the key players from a technical perspective (e.g. technical project leaders offshore and onsite, developers).

Developing the technical solution comprises implementation and testing. An approval step has to be passed again, this time internally by the offshore company's quality assurance staff. Afterwards, the approved solution is delivered to the offshore company's business consultant who wrote the functional specification. If the solution also passes this review step, the system is handed over to the customer for reviewing the final result with respect to the business problem and requirements that initiated the project.

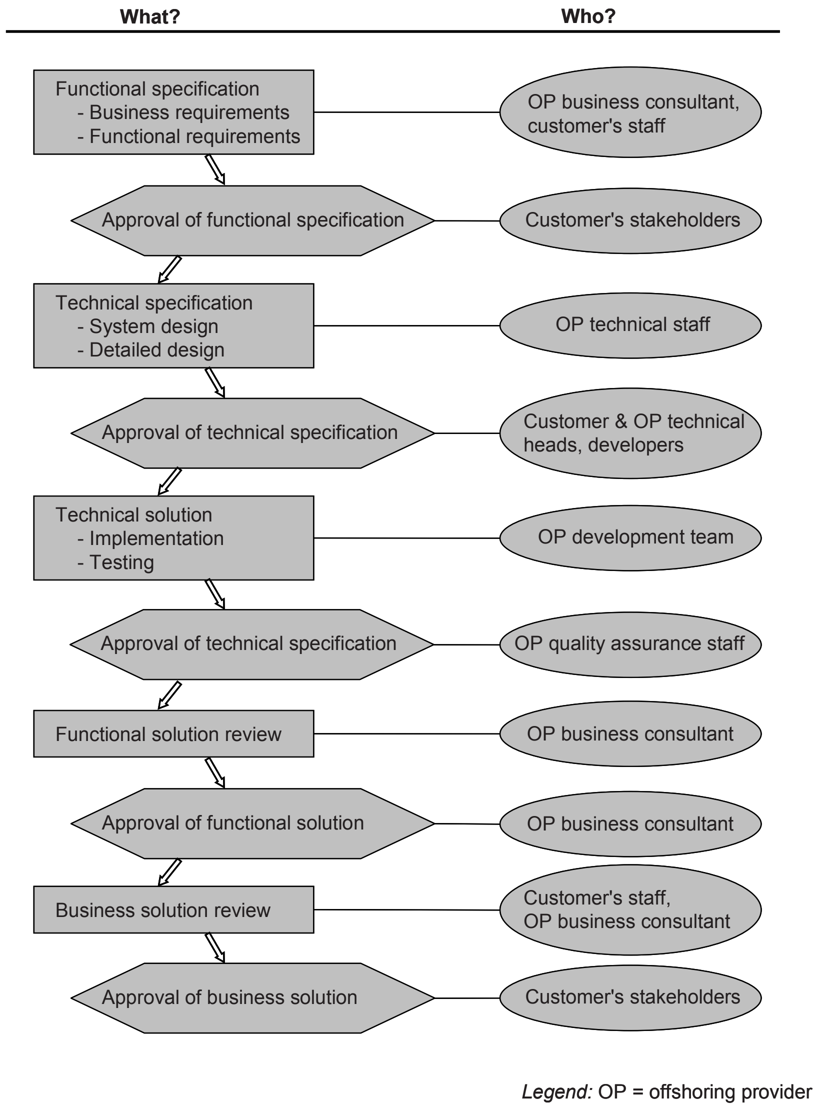
Generalized offshoring process model

Since offshoring projects differ in what tasks are outsourced to offshore, it is hardly possible to formulate a universal process model for all offshore ISD projects. Figure 4-26 attempts to capture the major stages that most such projects go through. Depending on the life-cycle stage where the offshoring starts, *offshoring-specific tasks* are entered earlier or later in the project – following either the business-problem specification, requirements analysis and definition, high-level design or detailed-design stage. Offshoring-specific stages are the following:

Figure 4-24 Criteria to examine for offshoring projects

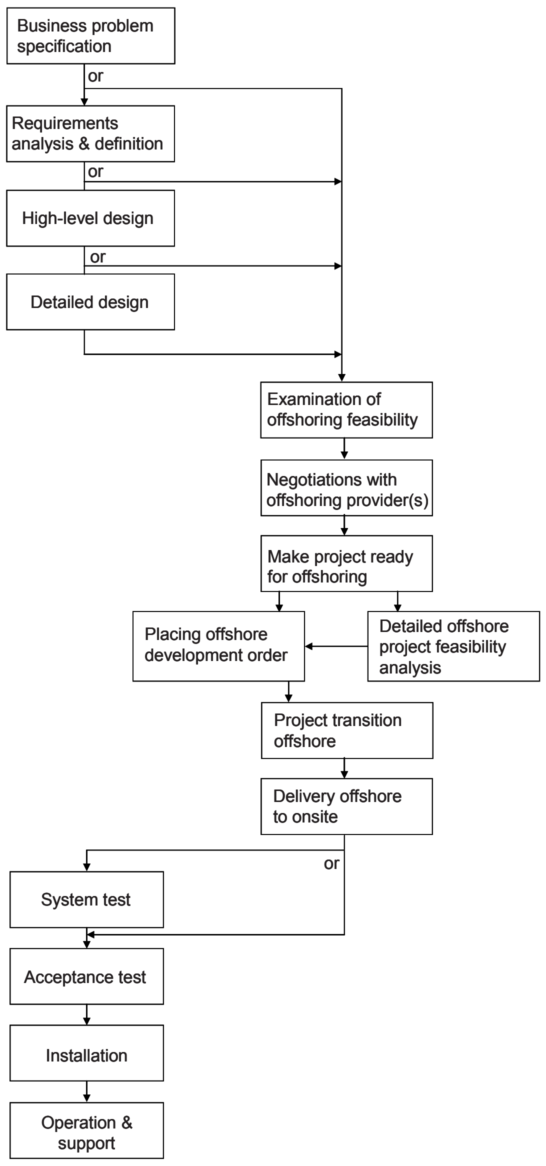
Criteria for Offshoring			
Is the management committed to offshore the project?	opposed	↔	strongly in favor
If the system under consideration is developed for a customer, does the customer agree to offshoring?	yes –	doesn't care	– no
What is the project size (in €, $, or person months)?	very small	↔	very large
Can the system easily be divided into separate modules, tasks and/or activities?	very easy	↔	very difficult
What is the share of standard technologies required in the project?	very low	↔	very high
Is the system tightly connected with other systems at the customer's site (i.e. how many interfaces with other systems)?	very loose	↔	very tight
How much knowledge of the underlying business process must the offshore provider have?	very little	↔	very much
How well is the system documented (e.g. completeness, correctness and understandability of specifications)?	very well	↔	insufficient
If there is a language barrier between the offshore and the onsite teams, can appropriate communication be ensured?	very easy	↔	very difficult

Figure 4-25 Offshoring project approval steps [DCM 2006]


What?
Who?
Functional specification
- Business requirements
- Functional requirements
OP business consultant, customer's staff
Approval of functional specification
Customer's stakeholders
Technical specification
- System design
- Detailed design
OP technical staff
Approval of technical specification
Customer & OP technical heads, developers
Technical solution
- Implementation
- Testing
OP development team
Approval of technical specification
OP quality assurance staff
Functional solution review
OP business consultant
Approval of functional solution
OP business consultant
Business solution review
Customer's staff, OP business consultant
Approval of business solution
Customer's stakeholders
Legend: OP = offshoring provider

- *Examination of offshoring feasibility:* The first offshoring-specific task, provided that offshoring is considered a serious option, is to investigate in more detail if the project is suited to offshoring. Criteria like the ones listed in figure 4-24 will be applied.
- *Negotiations with offshoring provider(s):* Requests for quotations are sought from the potential offshoring provider or from several providers, and negotiations with the provider(s) are conducted. This is a crucial stage if the provider is a different company. In the case of a captive center it is also important, because the cost of the project will be estimated so that the economy of the deal can be assessed.
- *Preparing the project for offshoring* includes the following tasks: Determining what has to be done before a work order can be placed with the provider; setting up a project organization that takes offshoring-specific requirements into account; establishing a rough overall project schedule and a more detailed transition plan; management commitment to outsource project stages offshore; and preparing onsite project members and other stakeholders to deal with the offshoring situation.
- *Detailed offshore project feasibility:* If necessary, the client and the offshore organization analyze in detail if it is reasonable to assume that the outsourced tasks will be solved as expected. The offshoring provider may need to collect information necessary to examine whether the required expertise, manpower and technical infrastructure are available or can be allocated at the offshore site in order to be able to make a definite commitment. The client's objective is to be convinced that the partner is reliable and capable of delivering as expected. This stage may include refining the project organization, the transition plan and the project schedule.
- *Placing the offshore development order:* Provided that the offshore organization is willing and capable of performing the outsourced tasks, both parties enter into an agreement specifying the work to be done and perhaps some process characteristics. While the legal form of such an agreement depends on whether the partners belong to the same company or not (e.g. a contract, a statement of work), some essential contents will be the same: timetable, milestones, deliverables, costs etc.
- *Project transition:* The major activities in this stage are knowledge transfer and ensuring a working project-wide technical infrastructure. The objective of this stage is to make sure that the offshore organization is able to continue the project successfully offshore.

Figure 4-26 A generalized offshoring life-cycle model


Business problem specification
or
Requirements analysis & definition
or
High-level design
or
Detailed design
Examination of offshoring feasibility
Negotiations with offshoring provider(s)
Make project ready for offshoring
Placing offshore development order
Detailed offshore project feasibility analysis
Project transition offshore
Delivery offshore to onsite
or
System test
Acceptance test
Installation
Operation & support

In order to acquire necessary project knowledge (e.g. domain, business-process, tool or environment knowledge), offshore personnel may have to be trained by the customer. For this purpose, employees of the offshore organization may visit the client's organization onsite to collect information and knowledge that they can take home to disseminate among their project co-workers.

– *Delivery:* While a significant portion of the workload will be carried out by the offshore organization, the next stage from the client's point of view is when will they get the results, i.e. the functioning software system. In addition to the software, the system documentation, testing results and quality assurance reports will be handed over to the client. Delivery usually takes place onsite, with offshore personnel available onsite to solve problems detected directly.

Depending on what tasks and stages were outsourced, the process continues at the customer's site with system or acceptance testing.

Offshoring-specific stages not necessarily strictly sequential

Although the model in figure 4-26 is basically sequential, with some activities possibly going on in parallel, the offshoring-specific stages need not necessarily be performed in a strict sequence nor as disjoint stages. For example, RFCs and negotiations can be performed before, after or parallel to making the project ready for offshoring. Likewise, if the offshore organization is already known, then the project feasibility study, leading to a final commitment to offshore the project, can be done together with the preparation for project offshoring.

Break points in the process

Furthermore, there are breakpoints in the process which are not explicitly marked in the figure. Such a point where the project may be cancelled or the process may go back to an earlier stage is, for example, the end of the offshore project feasibility analysis. If the offshore partner cannot credibly assure that the project is in good hands, the customer may negotiate with a different provider, or the offshoring idea may be completely dropped because the problems occurring with this provider are presumed to be the same as with other providers.

4.5.2 An Offshore Software Company's Perspective

Offshore software companies

Unless the offshoring provider was set up as a subsidiary or as a multinational’s pure captive center, the offshoring provider is a normal software company in its home country, perhaps specialized in working with foreign customers. From the point of view of such a company, an offshoring project is just another project that has to be acquired on the market, bidding against competitors. As software companies, offshoring providers offer more than just coding. Projects covering more stages or even the entire life cycle are more attractive to them than plain implementation or maintenance projects.

Process model: the earlier, the more freedom

With regard to the topic of this chapter of the book (i.e. process models) offshoring providers are restricted by the overall model imposed by the customer. The majority of process models found in practice are sequential models (waterfall model). However, the further up the system life cycle that an offshoring provider enters the process, the more freedom they have to form the process according to their needs and experience. For example, when the entire system development process is in the hands of the offshoring provider, this organization is free to choose an iterative or evolutionary approach. If the client is ready for continuous engagement and collaboration, even a comprehensive iterative approach such as RUP (cf. section 4.3) may be applied.

Initially: maintenance and coding & testing projects

Initially, the majority of offshore projects were maintenance and coding and testing projects, because offshore programmers were cheaper. Another focus was system operation. Typical transaction systems such as computer reservation systems were run offshore. As many organizations have accumulated offshoring experience over the years, the level of software-capabilities maturity has also grown. Due to this, offshore organizations have become trusted partners for pre-coding stages as well as for outsourcing entire business functions or processes.

Nowadays: precoding stages and entire processes

The more projects an offshoring provider successfully completes with the same customer, the closer the business relationship with that customer becomes, and the more likely it is that they will be awarded future contracts. With a long-standing relationship, *trust* between the partners grows, and the customer may be increasingly willing to outsource more critical process stages such as requirements analysis or even the entire solution process for the business problem to the offshore partner.

Many offshore companies offer a full spectrum of IT services

Many offshoring providers offer a full spectrum of IT services to their customers. As business organizations, they naturally attempt to sell those services with which they can generate most revenue. This means that instead of offering just coding and testing services, they are designing systems, analyzing and specifying the customer's requirements, and capturing and modeling the business problem. Along with business consulting services, globally operating offshore companies have become serious competitors in the western domestic consultancy industry.

TCS and Infosys are the world market leaders in offshore services

The world market leaders in the offshoring market, *Tata Consultancy Services (TCS)* and *Infosys Technologies*, are large Indian organizations that have been providing offshoring services for a long time. TCS is an IT company employing more than 104,000 employees worldwide (in 2007), certified at CMMI level 5 [TCS 2007]. Application development is one branch in their IT services division, next to business process outsourcing, consulting, infrastructure services, engineering and industrial services etc. Infosys Technologies with over 80,000 employees worldwide has been focusing on strategic offshore outsourcing of software services for many years [Infosys 2007]. Both TCS and Infosys use a global delivery model (GDM) for their services. Offshoring projects are embedded in these frameworks.

Global IT players vs. conventional offshoring providers

Large organizations like TCS and Infosys offer a fully-fledged spectrum of services and products beyond what is traditionally called "offshoring". They have their own proven process models, applied in many projects, covering the entire life cycle or major parts of it. On the other hand, a large number of small and medium-size offshoring providers worldwide still live largely from "traditional" offshoring projects in which the customer outsources implementation and testing or maintenance and imposes the overall process model.

5 Analysis and Design

Core activities in all process models

No matter which process model is followed in the project and no matter who does the work, certain activities will always occur. These activities may be conducted in a linear sequence (as in the waterfall model), in iterations (as in RUP) or in an evolutionary manner, yet in any case they have to be done. Core activities include the following:

- Requirements engineering
- Design
- Implementation
- Test

In chapters 5 and 6, approaches to solve the underlying problems and tools supporting the respective tasks are presented. The problems of requirements engineering, design, implementation and test have been there as long as information systems development has existed. Therefore the state-of-the-art regarding methodological approaches is rather stable. On the other hand, tools supporting the activities are subject to continuous improvement and change more rapidly.

5.1 Requirements Engineering (RE)

Objective: elaborating and documenting the requirements

The main objective of requirements engineering is to elaborate the requirements for the information system under consideration and to document them in an appropriate way, for example in a requirements specification as discussed in section 3.2.1. In order to achieve these results, requirements engineers need a thorough understanding of the problem and of the system to be built, involving identification of the tasks, the functionalities, the users and other stakeholders, the scope and the resources needed for building the system. Requirements engineering can be defined as follows:

Definition: requirements engineering

> *Requirements engineering (RE)* comprises all necessary activities of the IS development process that ascertain the stakeholders' requirements, analyze and evaluate these requirements, and document them so that they can be used in further development stages and throughout the information system's lifetime.

Requirements engineering thus comprises three major areas: *requirements elicitation* (ascertaining the requirements), *requirements evaluation* (analysis, agreeing, validation of the requirements) and *requirements specification* (documenting the requirements). Before the requirements elicitation starts, a *feasibility study* may be conducted.

Requirements management

When the information system is completed and in operation, or even when the system is still under development, *requirements management* is an issue. New requirements may emerge, asking for implementation in future versions of the system. Gathering requirements, prioritizing, monitoring and keeping track of the status (which requirements have been implemented and to what extent?) are activities that accompany the information system throughout its life cycle.

Requirements engineering emerged in the 1980's

Requirements engineering emerged as an important area for research and practice in the 1980's, initiating an academic discussion whether it should be considered as a discipline of its own or as a part of software engineering. No matter which position the contributors to the discussion assumed, all agreed on the importance of thoroughly engineering the stakeholders' requirements. As software systems became increasingly complex, users became less and less satisfied because they felt that their "true" requirements were not being met. Something had to be done and the birth of requirements engineering was the outcome.

5.1.1 What are Requirements?

What should the future system do and how should it behave?

Requirements are primarily descriptions of what the future system should do and how it should behave once it is implemented. Requirements may specify properties of or constraints upon both the system and the development process.

Requirements can be subdivided into functional requirements and non-functional requirements on the one hand, and into user requirements and system requirements on the other hand. Figure 5-1 illustrates these categories.

Functional requirements

– *Functional requirements* specify what the system should do – exactly what business problems the system should solve, what functions it should provide to the user, what screens, forms, reports, and data are needed, etc.

Non-functional requirements

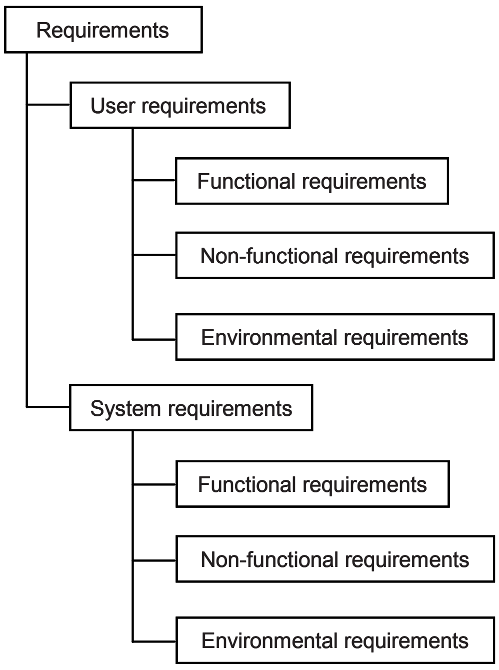
– *Non-functional requirements* are general criteria that the system should meet, apart from solving specific application problems. Such criteria often refer to software quality and performance of the system. Examples are maintainability, reliability, scalability, robustness, user-friendliness, response times, interoperability, reusability, modularity and understandability. These criteria have been extensively discussed for many years in the software engineering literature as attributes of software quality [ACM 1978, Boehm 1978, Kan 2002].

Environmental requirements

– Some requirements may be imposed by the legal environment, by industry standards the organization desires to meet, or by other influencing factors that are not directly related with the specific information system. Since requirements of this type may apply to an entire application domain, Sommerville calls them domain requirements [Sommerville 2007, pp. 125-126]. A more general term is *environ-*

mental requirements. An example is requirements derived from the W3C's accessibility guidelines for web pages [W3C 2006].

Figure 5-1 **Classification of requirements**


Requirements
User requirements
Functional requirements
Non-functional requirements
Environmental requirements
System requirements
Functional requirements
Non-functional requirements
Environmental requirements

Requirements are often specified on at least two different levels of abstraction, on the user's level and on the developers' level, leading to a distinction between user requirements and system requirements [Sommerville 2007, pp. 127-131]:

User requirements

– *User requirements* describe functional and non-functional requirements in a way that is understandable for system users and other stakeholders without detailed technical knowledge. They should only specify the external behavior of the system and not system characteristics relevant for the design or implementation of the system. The specification should be written in non-technical language (usually in natural language) using easy-to-understand diagrams.

- *System requirements* describe the requirements on a technical level and in more detail, explaining how the desired user functionality should be provided by the system. System requirements serve as the starting point for the system design. If the development of the information system is outsourced to a software vendor, the system requirements specification may serve as the basis for the contract. Therefore it should be a complete and consistent specification of the whole system.

System requirements

5.1.2 Major Tasks of Requirements Engineering

The requirements engineering process often starts with a feasibility study and then goes through the steps of requirements elicitation, analysis, agreeing (negotiation), validation and specification. Figure 5-2 illustrates how the tasks of requirements engineering work together.

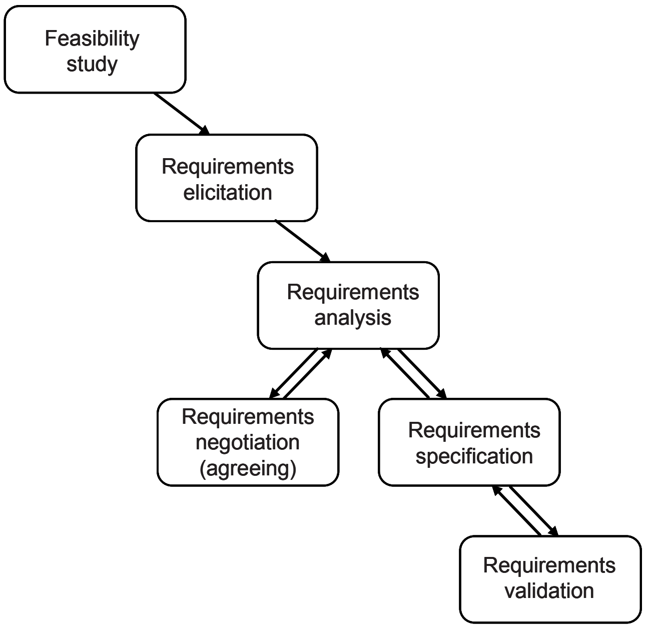
Feasibility study

Updating and refining earlier examinations

Depending on how detailed the original feasibility study of the system was from when the project proposal was written (see sections 3.1 and 3.2), the *feasibility study* will update and elaborate the previously examined points in detail. Questions to be answered include the following:

- Is the system economically feasible, i.e. can it be built within the given budget?
- Is the system technically feasible, i.e. can it be built with the available software and hardware technology? Can required additional technology be bought within the project budget?
- Are the required human resources with know-how and experience for the project available?
- Is the system organizationally feasible, i.e. will the organization adopt the system successfully? Is the system legally feasible, i.e. does it comply with all laws and legal regulations that apply?
- Can the system be built within the given time frame?

Figure 5-2 **Requirements engineering tasks**



Requirements elicitation

Latent needs have to be "elicited"

Eliciting requirements is also called capturing or gathering requirements. Eliciting is the preferred term because it describes that something latent needs to be brought out. In many cases, requirements are neither explicitly available nor completely clear in the minds of the stakeholders, so they definitely need to be "elicited". At the same time, analysts must develop their understanding of the application domain.

Requirements elicitation comprises the following parts [Nuseibeh 2000, p. 37]:

System boundaries

– Finding out what problem needs to be solved, and hence identifying *system boundaries*. These boundaries define, at a high level, where the final system will fit into the current operational environment.

- Identifying *stakeholders* – individuals or organizations who will gain or lose from the success or failure of a system. Stakeholders include customers or clients who pay for the system, developers who design, construct and maintain the system, and users who interact with the system due to their work. Users may belong to different categories, e.g. clerical workers, knowledge workers, managers. They can be novice users, expert users, occasional users, disabled users etc. An essential part of the elicitation process is to identify the needs of different user classes.

Identifying stakeholders

- Eliciting high-level *goals* or objectives a system must meet. This helps to focus the requirements engineer on the problem domain and the needs of the stakeholders, rather than on possible solutions to those problems.

High-level goals or objectives

- Eliciting information about the *tasks* users currently perform and those that they might want to perform with the help of the new system. This is the part of requirements elicitation that will finally be transformed into operational functional requirements specifying what the system should do, with an appropriate level of detail.

How the tasks are currently performed

A number of *techniques* are available for the elicitation of requirements. An overview can be found in Nuseibeh and Easterbrook's "roadmap" [Nuseibeh 2000, p. 39]. The most common techniques are the use of questionnaires and surveys, interviews and analysis of existing documentation such as organizational charts, process models or standards, and user or other manuals of existing systems.

Techniques for requirements elicitation

Group elicitation techniques include brainstorming and focus groups, as well as RAD (rapid application development) and JAD (joint application development) workshops. Such workshops bring analysts, developers, customers and other stakeholders together with the help of an unbiased facilitator.

Prototyping as discussed in section 4.2.2 is a useful approach when there is a great deal of uncertainty about the requirements and what the final system should be like, or when early feedback from stakeholders is desired. Prototypes may be developed for requirements elicitation only and then discarded (throw-away prototyping).

Throw-away prototyping

Other techniques for requirements elicitation are model-driven techniques and cognitive techniques such as protocol analysis, laddering, card sorting and repertory grids [Nuseibeh 2000, p. 39]. Sommerville proposes viewpoint-oriented elicitation, scenarios and ethnography as elicitation techniques [Sommerville 2007, pp. 149-158].

Requirements evaluation

Analyzing the requirements

Once the requirements have been collected, the next step is to analyze them. *Requirements analysis* begins with a classification in which the analyst takes the unstructured requirements and groups them into coherent clusters. Requirements may have to be prioritized due to budget or schedule limitations. Some requirements may be incompatible as they were articulated by different stakeholders expressing their views independently of each other. This may have resulted in conflicting, redundant or overlapping requirements, as different stakeholders have different goals, which is another source of conflicts.

Agreeing on final requirements

An important part of the requirements evaluation stage is therefore to come to an *agreement* about the final requirements to pursue and their priorities. This usually involves negotiations among and decisions by the stakeholders. Requirements negotiation and conflict resolution is a difficult step because of different opinions, background, knowledge and last but not least due to the different goals of the stakeholders.

Requirements validation

Requirements validation is the step in which the requirements are formally examined and approved. While the checking of requirements also happens in analysis, Sommerville uses the term validation to describe a more formal way of checking requirements which is based on written documents or models that will be discussed below [Sommerville 2007, pp. 158-160]. These checks include checking for consistency, completeness and realism (i.e. can the requirements reasonably be implemented?). Special review teams may be assigned to this task.

The validation process can be facilitated through system prototypes demonstrating certain system behavior. Writing test cases as part of the validation process can also help to detect errors in the requirements. If it is difficult or impossible to specify how the realization of a particular requirement can be tested, then the requirement might not lend itself straightforwardly to implementation in the first place and should be revised.

Requirements specification

Formal documents are needed

The step in which the requirements are formally documented is the requirements specification or documentation. Formal documents are needed for various purposes, as discussed in section 3.2.1. Figure 2-4 showed an outline of a requirements specification document.

Ideal requirements are "complete, consistent, correct, feasible, necessary, prioritized, unambiguous and verifiable" [Wiegers 2001]. An adequate specification of the requirements is important for the upcoming development activities, in particular for the design stage.

Semi-formal representations aid the designers

Some formalization may be preferred for this purpose, because formal or semi-formal representations are usually closer to the needs of the designer than plain written text. Modeling methods for requirements engineering are discussed in the next section.

Requirements engineers

We have referred to the persons responsible for the requirements engineering tasks in this section as *requirements engineers* or analysts. Other names are requirements managers, requirements analysts, business analysts and system analysts. Requirements engineers must be versatile persons, capable of working with the client or customer, with the end-users, the project sponsor and/or the product manager. This is a quite demanding role.

Skills a requirements engineer should have

Serving as "... the principal conduit through which requirements flow between the customer community and the software development team", the requirements engineer must posses an array of skills such as the following [Wiegers 2003]:

» Listening skills – understanding what people say and what they might be hesitant to say
» Interviewing and questioning skills – asking the right questions to elicit essential requirements information
» Analytical skills – critically evaluating the information gathered from multiple sources to reconcile conflicts, separate user wants from needs, and distinguish solution ideas from requirements
» Facilitation skills – for example, leading requirements elicitation workshops
» Observational skills – being able to validate information and expose new areas for elicitation
» Writing skills – communicating information effectively to customers, marketing, managers and technical staff
» Organizational skills – structuring the many pieces of information gathered during elicitation and analysis into a coherent whole
» Modeling skills – being able to model requirements information and represent it in graphical diagrams or in a modeling language
» Interpersonal skills – negotiating with and resolving conflicts among project stakeholders

5.1.3 Use-case Modeling and Other RE Methods

The most common approach to modeling and representing functional requirements nowadays are *use cases.* They were introduced by Ivar Jacobson in 1986 and later became a part of UML. A use case is a textual description of what a system should do. Use cases are sometimes called *stories* as they explain how a system should fulfil the stakeholders' needs. The example shown in figure 5-3 describes a use case on a high level.

Figure 5-3 **A brief use case [Larmann 2005, p. 63]**

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Actors initiate actions

Actors are the ones that interact with the system. They carry out the use cases. An actor stands for a role rather than for an individual. A human being is not the only actor. A software system, a piece of hardware, or in general terms, something with behavior are also actors. They are the stimuli that initiate actions.

Scenarios (use case instances)

A specific sequence of actions and interactions between actors and the system is called a *scenario* or a *use-case instance*. A use case can thus be characterized as a set of scenarios describing actors using the system to obtain an observable result of value. Jacobson et al. define a use case as "a behaviourally related sequence of transactions performed by an actor in a dialogue with the system to provide some measurable value to the actor" [Jacobson 1995, p. 343.]

Success scenarios and failure scenarios

Some scenarios may be desirable, representing successful actions and interactions with the system, while others stand for failures. Both *success scenarios* and *failure scenarios* are described within the use case. In many cases, there is one main scenario (called the basic flow or *main success scenario*) and a number of *alternate scenarios* (or failure scenarios). The main success scenario is often the one assumed in the first high-level description of the use case.

Use-case model

A set of use cases, actors and their relationships is called a *use-case model*. Such a model comprises all the actors of the system and all the use cases by which the actors interact with the system. In this way, a use-case model describes the total functional requirements of the system.

Types of use-case descriptions

Use cases can be short informal descriptions or detailed specifications, depending on the purpose for which they are written. Three types are often distinguished [Larmann 2005, p. 66-67]:

– *Brief* – one paragraph with a few sentences summarizing the use case (see figure 5.3 above for an example).
– *Casual* – a few paragraphs of text, describing various scenarios in an informal manner (see figure 5-4 below for an example).
– *Detailed ("fully dressed")* – a formal document elaborated in detail with certain sections, including preconditions and postconditions for success. A full expansion of the casual use case of figure 5-4 can be found in [Larman 2005, pp. 68-72].

Fig. 5-4: A casual use case [Larman 2005, p. 63-64]

Handle Returns

Main success scenario

A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item. The system …

Alternate scenarios

If they paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting systems, …

Use-case templates

Detailed use cases are often based on templates. Such a template provides typical sections such as use-case name, summary of the use case, preconditions, basic flow, alternate flows, postconditions, author, date etc. A common use-case template from the author of "Writing Effective Use cases" [Cockburn 2000] with sample data is shown in figure 5-5.

Preconditions and postconditions

The *goal-in-context* section specifies what the major stakeholder (buyer) expects the system to do for him or her. *Preconditions* state what must be true before the scenario starts. *Success end conditions* (or *postconditions*) state what must be true when the use case is completed successfully (according to the main success scenario). *Failed end conditions* must be true when an alternate scenario was followed.

Primary and secondary actors

Primary actors are the ones whose goals are fulfilled through the use cases. *Secondary actor*s (or supporting actors) provide a service to the system. Secondary actors are often other information systems interfacing with the system under consideration. A *trigger* is an action upon the system that starts the use case. *Extensions* refer to the steps of the main scenario. *Sub-variations* will cause eventual bifurcations in the scenario. Since use cases can be composite (i.e. composed of other use cases), the *superordinate use case* is the use case that includes this one.

Use-case diagrams

UML provides use-case diagrams

Use-case advocates stress that use cases are text documents. "Doing use case work means to write text" [Larman 2005, p. 89]. Nevertheless, visual representations of use cases have become very popular. The UML (unified modeling language) provides a graphical notation to illustrate the relationships between actors and use cases, so called use-case diagrams.

Actors are usually represented by a stickman with the name written underneath. The suggestive meaning of the stickman symbol is a human being, but actors can be other computer systems as well. Therefore, a box-type representation of actors is often used, with the stereotype *«actor»* and the name of the actor written within the box. Using both types of representation – stickmen for human actors and boxes for computer-based actors – provides a visual distinction between the two types.

UML symbols of use cases

Ovals are the symbols for *use cases* in a use-case diagram. Actors and use cases are connected by lines showing which actor communicates with which use cases. The lines are called *communication associations*.

Figure 5-5 **Detailed use case based on a template [Cockburn 1998]**

Use Case 5	Buy goods	
Goal in context	Buyer issues request directly to our company, expects goods shipped and to be billed.	
Scope & level	Company, summary	
Preconditions	We know buyer, their address, etc.	
Success end condition	Buyer has goods, we have money for the goods.	
Failed end condition	We have not sent the goods, buyer has not spent the money.	
Primary, secondary actors	Buyer, any agent (or computer) acting for the customer. Credit card company, bank, shipping service.	
Trigger	Purchase request comes in.	
Description	Step	Action
	1	Buyer calls in with a purchase request.
	2	Company captures buyer's name, address, requested goods, etc.
	3	Company gives buyer information on goods, prices, delivery dates, etc.
	4	Buyer signs for order.
	5	Company creates order, ships order to buyer.
	6	Company ships invoice to buyer.
	7	Buyer pays invoice.
Extensions	Step	Branching action
	3a	Company is out of one of the ordered items: 3a1. Renegotiate order.
	4a	Buyer pays directly with credit card: 4a1. Take payment by credit card (use case 44)
	7a	Buyer returns goods: 7a. Handle returned goods (use case 105)
Sub-variations		Branching action
	1	Buyer may use phone in, fax in, use web order form, electronic interchange
	7	Buyer may pay by cash or money order, check, credit card
Related information	5. Buy goods	
Priority	Top	
Performance	5 minutes for order, 45 days until paid	
Frequency	200/day	
Channel to actors	Not yet determined	
Open issues	What if we have part of the order? What if credit card is stolen?	
Due date	Release 1.0	
...any other management information...		
Superordinates	Manage customer relationship (use case 2)	
Subordinates	Create order (use case 15) Take payment by credit card (use case 44)	

UML provides use-case diagrams

Figure 5-6 illustrates this graphical notation with an example of a high-level use-case diagram for a POS (point-of-sale) system [Larman 2005, p. 90]. Note that the use cases "process sale" and "handle returns" from the figures 5-3 and 5-4 are contained in the diagram. Primary actors are the "cashier", the "system administrator" and the "sales activity system". Supporting actors in this example are software systems, namely the "payment authorization service", "tax calculator", "accounting system" and "HR system". As a matter of style, primary actors are placed on the left-hand side of the diagram and secondary actors on the right-hand side.

High-level use-case diagrams serve as context diagrams

High-level use-case diagrams like the one in figure 5-6 serve as *context diagrams* as in other diagramming notations for requirements engineering (see below). A context diagram shows the main behavior of the system under consideration and which actors communicate with the system. It makes visible what is inside and what is outside the system, i.e. it shows the boundaries of the system.

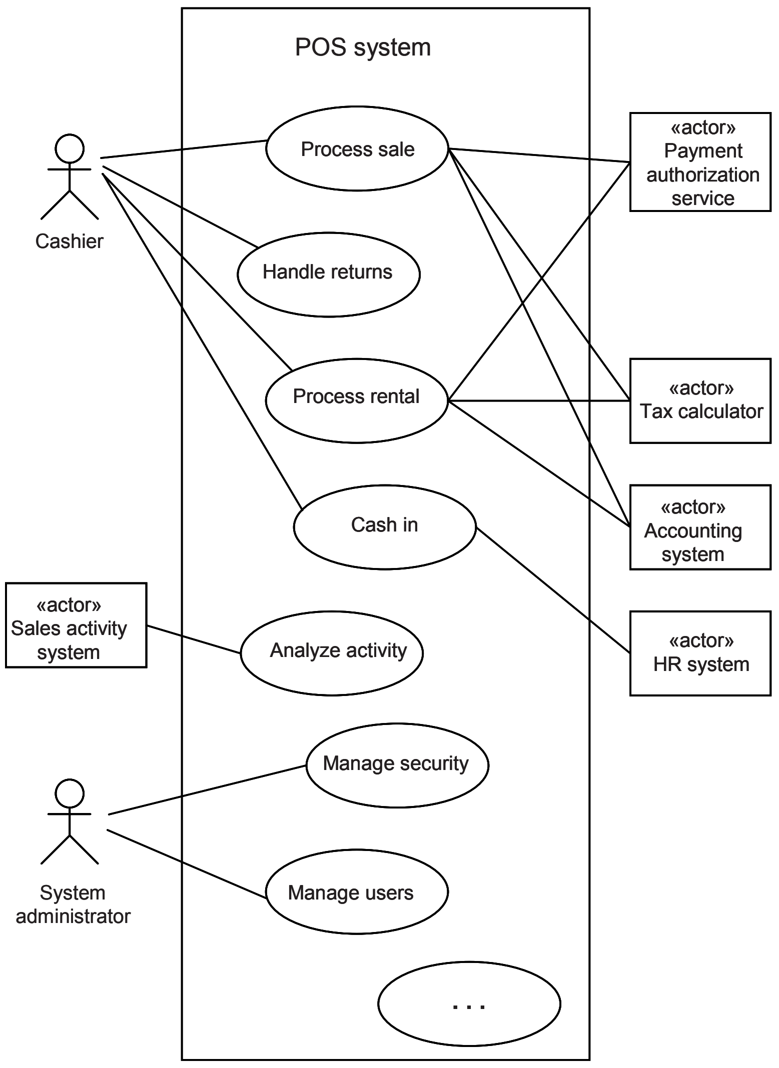
Relationships

Relationships between use cases

Use-case models can be refined in various ways. More structural information can be added by specifying relationships between use cases and relationships between actors. The relationship types used for this purpose reflect typical object-oriented concepts such as generalization and inheritance. Relationships between *use cases* are:

- *«Extends»* – a generalization/specialization relationship between two use cases. Use case B extends use case A with additional behavior which A does not have. Figure 5-7 a) illustrates this case. Processing CD sales might require more actions than processing normal sales. The cashier has to take action to get the CD that belongs to the empty CD cover the customer took from the shelf.
- *«Includes»* – a relationship of one use case that includes (or uses) the behavior of another use case. Use case A including use case B means that extra behavior from B is added to A. «Includes» is used when multiple use cases have a common function that can be used by all. Figure 5-7 b) shows an example. Returns at the POS require reimbursement to the customer, with a certain procedure for returning cash and a different one for returning the amount to the customer's credit card account. Returns by mail require the same procedure in case the customer paid by credit card.

Figure 5-6 **Use-case diagram for a point-of-sale system**[§]



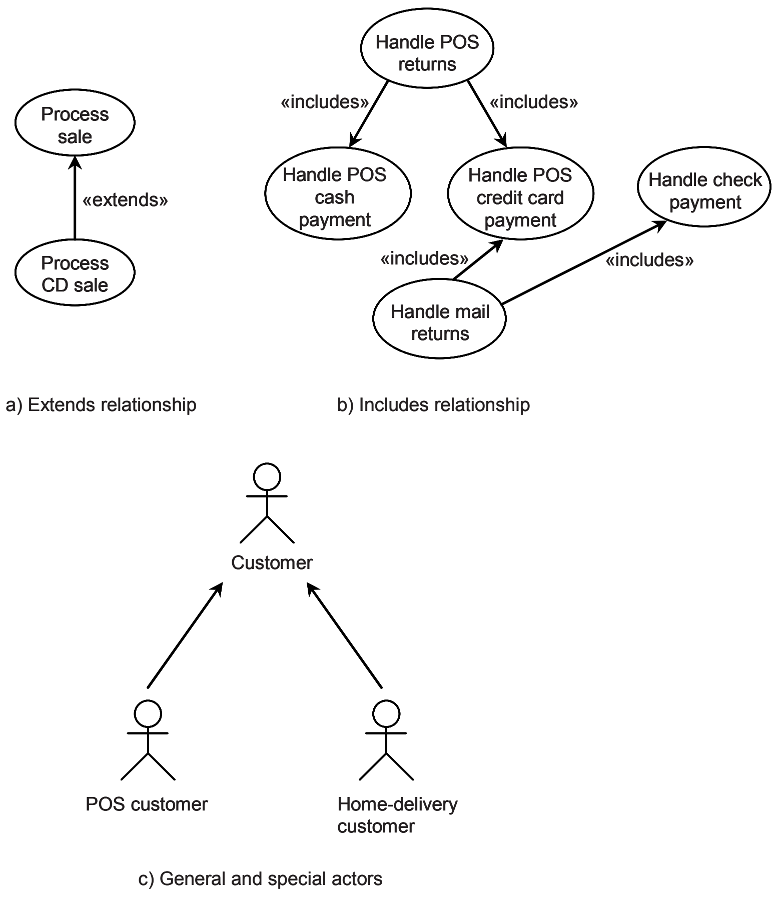
§ Larman 2005, p. 90.

A relationship between actors is:

Generalization - specialization

- *Generalization/specialization* – a relationship in which one actor inherits behavior from another one. Actor B is a specialization of A if B has some features inherited from A. In the example of figure 5-7 c), customers are specialized into customers at the point of sale and customers receiving home delivery.

Figure 5-7 Relationships in use-case diagrams (examples)



Supplementary specification

Feature lists

For a more detailed specification of requirements, additional documents supplementing the use-case model can be used. Feature lists are sometimes used for this purpose. While high-level feature lists contain essentially the same information as a use-case model, feature lists can be refined to as much detail as one likes. In this way, feature lists tend to become very long, obscuring the essential behavior that should actually be specified.

Supplementary specification

While use cases are helpful for capturing and documenting functional requirements, they are not suited to specifying non-functional requirements. The latter ones are often summarized in a *supplementary specification*, including [Larman 2005, p. 107]:

– Quality attributes and common functionality across many use cases – so-called FURPS+ requirements (functionality, usability, reliability, performance, supportability)
– Reports, documentation (user, installation, administration) and help features
– Hardware and software constraints (e.g. operating system, networking software)
– Development constraints (e.g. tools, IDE to use)
– Licensing, legal and internationalization concerns
– Packaging
– Standards (technical, safety, quality)
– Operational concerns (e.g. frequency of backups, handling errors)
– Information in the domain of interest (e.g. what is the entire cycle of credit payment handling)

When the requirements specification is intended to serve as a basis for a contract with a third party or as input for the design group, more detailed information than provided in the initial use-case model is needed.

In the *RUP (*Rational unified process; cf. section 4.3) model, this information is added throughout the process. Requirements engineering in RUP is an activity that is repeated in the iterations of the inception, elaboration and construction phases. In this way, the requirements engineer can start with the *inception* by identifying the most use cases, but writing only a few important ones in detail. In the *elaboration* iterations, the initial use cases are revised, and most use cases are eventually specified in detail. During *construction*, use cases may still be modified according to feedback received from implementation.

5.1.4 More UML: Sequence Diagrams and Class Diagrams

In addition to use-case models, high-level sequence diagrams and class diagrams are used in requirements engineering in order to refine the information captured in use cases.

Sequence diagrams

Sequence diagrams show the interaction between objects

A *sequence diagram* generally illustrates the interaction between objects. When used to specify high-level system behavior, a sequence diagram shows the communication between actors and the system under consideration for a particular scenario as a sequence of events or communication steps.

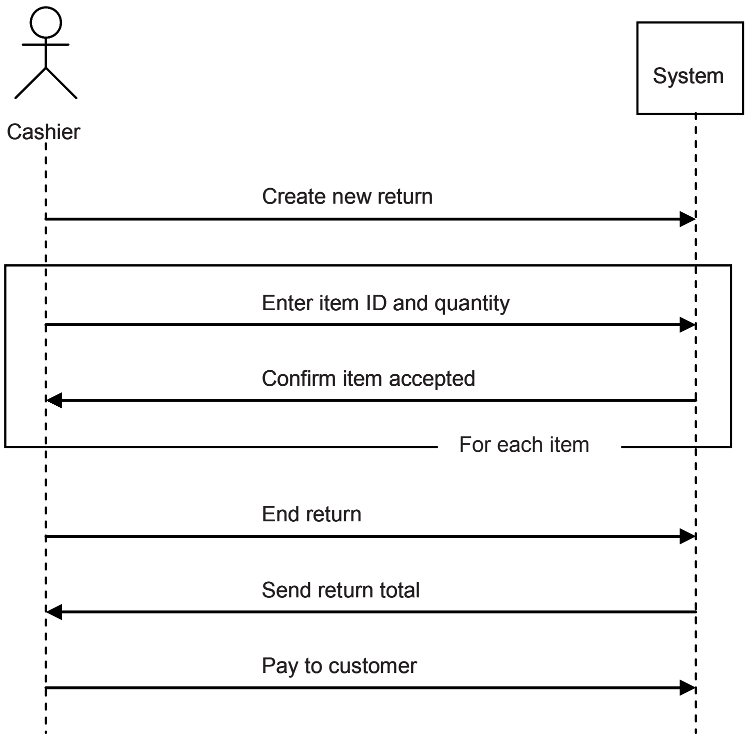
On top of the diagram are the objects involved – the actor(s) and the system. Vertical dashed lines indicate the beginning or the end of a communication between the two. Messages from the actor requesting an action by the system and returning messages from the system, including results, are represented by horizontal arrows. The sequence of actions is from top to bottom, i.e. the imaginary vertical axis represents the time.

In a high-level sequence diagram, the system is a black box

Figure 5-8 shows an example of a sequence diagram for the main success scenario of the "handle cash payment" use case in figure 5-7. The box around the arrows "enter item ID and quantity" and "confirm item accepted" indicates a loop – in case the customer returns more than one item. High-level sequence diagrams like the one in figure 5-8 regard the system under consideration as a black box, specifying only how the external actors communicate with the system.

Sequence diagrams are usually created for the more important scenarios, not for all. This diagramming technique is also used for later activities of information systems development, especially for system design. Here they serve the purpose of specifying object interaction through messages. This will be described in section 5.2.2.

Figure 5-8 **Sequence diagram example**



Class diagrams

A class diagram represents a static model of a system in terms of classes, attributes and relationships among the classes. The terminology for class diagrams, just as most of the UML terminology, is object-oriented. Note, however, that use cases and use-case models are not object-oriented.

Classes, attributes and relationships

A *class* is an object-oriented concept that defines a type of object. Objects belonging to the class share a common structure and a common behavior. Class diagrams can be used on different levels of abstraction. High-level class diagrams are called *domain models* (in RUP) or

High-level class diagrams

conceptual models. These models contain conceptual classes (or domain concepts) and their relationships.

Conceptual classes

A *conceptual class* is a type of object that occurs in the domain and is considered important enough to be represented on its own. It represents abstract or concrete things in the real world. In the above POS system, examples are "sale", "article", "customer", "cash register" etc.; or in a college administration system, "student", "course", "lecturer", "lecture hall" etc. In contrast to these high-level classes that represent domain concepts, lower-level classes used in design and implementation are software entities.

Finding appropriate conceptual classes is a creative activity. A common technique is to examine the use-case model and identify the nouns in the use cases' textual descriptions. The nouns are often reasonable candidates for conceptual classes. (In entity-relationship modeling, the same technique has been used for a long time. Finding appropriate entity types is a similar problem as finding conceptual classes.)

Relationships

In addition to classes, a conceptual model specifies the *relationships* between the classes or between their objects. Relationships considered in the domain model are association, generalization and refinement. (More relationship types will be introduced later for the design and implementation activities.)

Associations

An *association* is a semantic connection between classes, specifying that objects of these classes can be connected in a certain way. A particular connection of an object belonging to class A with an object of class B is called a *link*. An association can thus be described as a set of possible links between objects of the two classes. An association has:

– one or two names,
– one or two multiplicity expressions (optionally),
– one or two arrows (optionally).

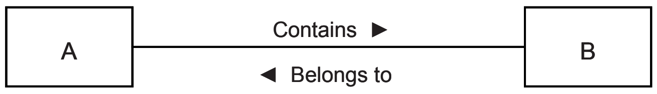
The *name* should express the meaning of the association (how are the objects connected?). An *arrow* is a visual aid to indicate in which direction to read the association. In figure 5-9, the association between A and B reads "A contains B" and "B belongs to A".

Multiplicities

Multiplicity specifies how many objects of class B can be associated with one object of class A. Typical values are:

1	one A object can be linked with exactly 1 B object
0..1	one A object can be linked with 0 or 1 B object(s)
0..*	one A object can be linked with 0 or more B objects
*	one A object can be linked with "many" (including 0) B objects; i.e. same as 0..*

Figure 5-9 **Association names and reading directions**



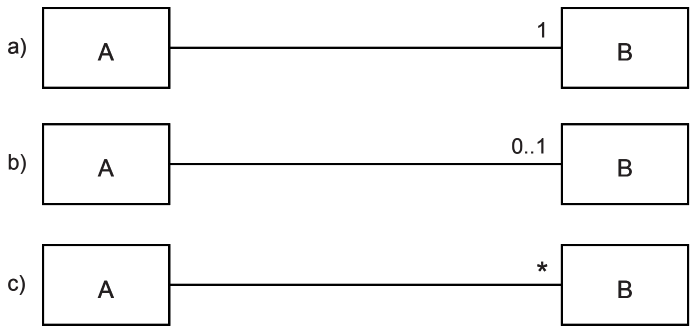
Other ranges than 0..1 (e.g. 1..4) and lists of values (e.g. 2, 4, 8) can also be used as multiplicities if the domain concepts asks for such associations. Figure 5-10 shows three cases:

a) An object of class A is linked with exactly 1 object of class B.
b) An object of class A is linked with at most 1 object of class B.
c) An object of class A can be linked with no, one or many objects of B.

Note that no multiplicity is specified for the reading direction from right to left. An assumption in such a case is that the multiplicity is one.

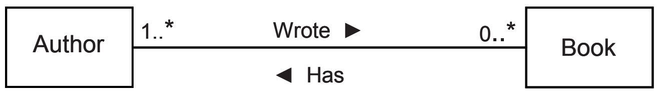
An example of multiplicities is given in figure 5-11. An author may have written many books, and a book can have more than one author.

Figure 5-10 **Association multiplicities**



Why is the multiplicity 0..* for "Wrote" and 1..* for "Has"? The answer is that a book without an author does not make sense, but an author without a book might. The publishing company might wish to record a promising author in its information system even before they have published the first book of this author.

Figure 5-11 **Association of books and authors**



Attributes

Attributes describe the characteristics of the objects belonging to a class. For example, an article has a name, an ID, a price etc. A sale has a total, a date and a time. In UML, attribute names are written in a second compartment of the box representing the class. In the domain model, only the essential attributes are considered.

Sometimes it depends on the perspective of the modeler to decide if something is an attribute or a class. An address, for example, could be considered an attribute if it is simple and belongs to only one person. A complex company address, on the other hand, qualifies as a class of its own if it is associated with several departments or contact persons in the company. Figure 5-12 shows a portion of a domain model with four conceptual classes, associations and some important attributes.

Attributes have certain types. In the high-level domain model, types are usually ignored, but in more detailed models the types are included. Types are either simple data types (such as integer, number, string, Boolean) or commonly used types (such as amount, phone number, or social security number).

Generalization

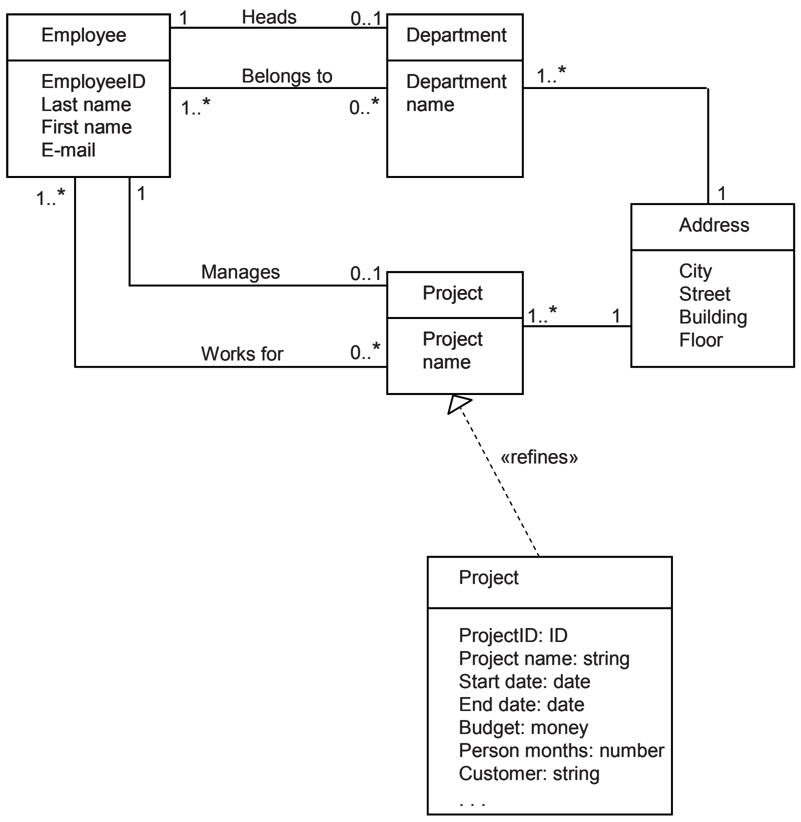
Generalization is a relationship between two classes indicating that one class, the superclass, is a more general form of the other class, the subclass; or in other words, the subclass is a special case of the superclass. We can also say that a subclass (object) *is a* superclass (object). For example, a supplier is a business partner. This is called an *is-a association*.

Superclasses and subclasses

The *superclass* is also known as the parent class. *Subclasses* are called child classes, children or derived classes. Generalization helps to reduce redundancy by factoring out common features of conceptual classes to a superclass.

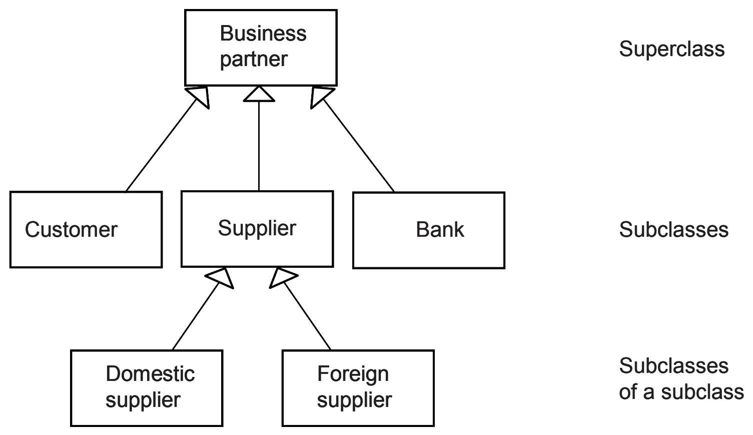
Figure 5-13 illustrates the *generalization* relationship with an example containing different types of business partners: customers, suppliers and banks. In UML, generalization is indicated by hollow arrowheads. All subclasses have common attributes, e.g. an address and a contact person, and common associations with other conceptual classes. These common features are factored out to the superclass "business partner". The features which are different remain with the subclasses.

Figure 5-12 **Classes, attributes and associations**



For example, suppliers may have an association with materials that can be purchased from them whereas banks do not. Therefore the superclass "business partner" will not have a general association with the materials class, but the subclass "supplier" will be associated with materials.

Figure 5-13 **Generalization example**



Generally speaking, all of a superclass's definition should apply to the subclasses. This means that the subclass has the same attributes, the same behavior and the same associations as the superclass. In addition, a subclass may have different features, otherwise there is little reason to create the subclass. Identifying subclasses and treating them separately is motivated by the need to capture additional attributes, additional associations or different behavior which is not provided by the superclass.

Generalization is transitive

Generalization is *transitive*. This means if C is a subclass of B and B is a subclass of A, then C is also a subclass of A with the same attributes, behavior and associations as A. In figure 5-13, "domestic supplier" is a subclass of "supplier" and indirectly of "business partner".

Refinement relationship

A *refinement* is a relationship connecting two descriptions of the same thing at different levels of abstraction. The refining description must conform to the abstract description. Refinements in UML can be used to model various matters. In requirements engineering, they are useful to refine conceptual classes of the high-level domain model into more detailed descriptions suited for a requirements specification document. Refinements are indicated by a dashed line with a hollow arrowhead. This is shown in the lower part of figure 5-12 where the class "project" is described in more detail.

5.1.5 Other Approaches to Requirements Engineering

While the most popular approach to requirements engineering today is to develop use cases and other UML models, more approaches exist, including structured analysis, business area analysis and ARIS requirements definitions.

Structured analysis (SA)

Structured analysis (SA) has been the dominating approach for many years and is still used, both in practice, research and teaching. In the 1980s and 1990s, most textbooks on information systems development were focused around structured analysis, discussing ISD as if structured analysis was the only approach available.

Ed Yourdon, Tom DeMarco, Chris Gane, Trish Sarson

SA was originally developed in the 1970s by Ed Yourdon, Tom DeMarco, Chris Gane and Trish Sarson. Many books by the original authors and by other others about SA have been written since then. SA is often used in combination with SD (structured design) and referred to as SA/SD.

Purpose of SA: creating a "structured specification"

The purpose of SA is to create a system specification. DeMarco starts his book "Structured analysis and system specification" with a crisp statement summarizing this purpose: "Let's get right to the point. ... Structured analysis is concerned with a new kind of functional specification, the structured specification" [DeMarco 1978, p. 3]. DeMarco's definition of SA is pragmatic: "Structured analysis is the use of these tools: data flow diagrams, data dictionary, structured English, decision tables, decision trees to build a new kind of target document, the structured specification." [DeMarco 1978, p. 16]

SA variants: DeMarco, SSA, modern structured analysis, SADT

The primary modeling tools of SA are *data flow diagrams (DFDs)*. They dominate SA to an extent that the terms "structured analysis" and "data flow diagrams" have even been used as synonyms. Another quasi-synomym is *DeMarco method,* although this is just one popular variant of SA. Others are SSA (structured systems analysis [Gane 1979]), modern structured analysis [Yourdon 1989] and SADT (structured analysis and design technique [Ross 1977]).

The structured specification is a graphical model of the processes

The *structured specification* is primarily a graphical model of the processes of the information system under consideration, represented by a set of data flow diagrams. A process in SA is an activity that transforms input data flows into output data flows. A data flow diagram is thus a procedural description of how activities are connected by data flows.

DFDs are created in a top-down manner, by functional decomposition. The structured specification comprises a data dictionary documenting the processes, data flows, data stores (e.g. files) and data elements, and so-called *transform descriptions* specifying how the processes should work. Structured English, decisions tables and decision trees are used for this purpose [DeMarco 1978, pp. 31-32].

SA process model and documents

SA provides a process model to arrive at the structured specification. This model consists of seven steps, starting with a study of the current situation and ending with the structured specification. The documents produced in each step are called:

1. *Current physical data flow diagram* – result of a study of the current environment, in particular of the current way of solving the underlying problems (who does what and how?)
2. *Current logical data flow diagram* – result of a clean-up and abstraction process elaborating the current processes and data flows based on the current physical data flow diagram
3. *New logical data flow diagram* plus supporting documentation – a representation of the requirements for the new system in terms of processes and data flows, described in detail in the data dictionary and the transform descriptions (at this point it is not decided yet whether the processes will be automated or manual)
4. *New physical data flow diagrams* – a set of options resulting from considerations to determine the scope of the automated system, i.e. the question what will be automated and what will remain manual work ("establishing the man-machine interface")
5. *New physical data flow diagram* – the selected option based on quantification of cost and schedule data associated with each option
6. *Structured specification* – the target of the process, as a result of revising and packaging the new physical data flow diagram, the data dictionary and the transform descriptions into a final form

Context diagram defining the boundaries of the system

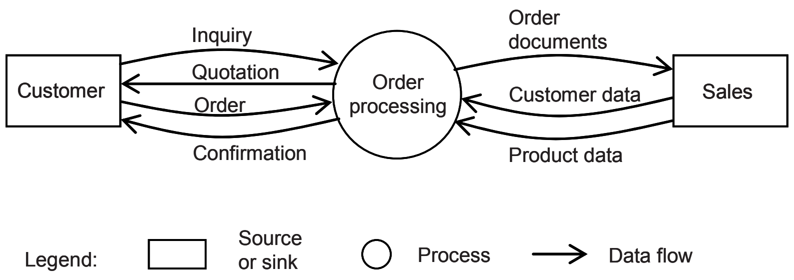
Developing the data flow diagrams is a top-down process with stepwise refinement. Higher-level diagrams are decomposed into lower-level diagrams. The domain of study is specified in a *context diagram* that defines the boundaries of the system. In a context diagram, system functionality is outlined as one top-level process with ingoing and outgoing data flows and possibly sources and sinks of the data. A *source* or *sink* is a person, an organization or an information system outside the context of the system under study, providing data for or receiving data from the system.

Figure 5-14 illustrates this concept with the help of a simplified order-processing system. Customer inquiries and possibly orders for products are coming from customers. The order-processing system issues quotations and order confirmations. For this purpose it needs product data and customer master data provided by the sales department (or sales information system). When the order is confirmed to the customer, order documents are issued and given to the sales department for further processing. "Customer" and "sales" are sources and sinks of data simultaneously because they both provide and receive data.

Level-0 diagram

On the highest level of decomposition (level 0), the main processes of the system and the data flows into and out of these processes are plotted. Such a *level-0 diagram* for the order-processing system is shown in figure 5-15. Note that the six processes are numbered and that "orders & quotations" is a so-called data store (e.g. a file or a database table) indicated by two parallel lines.

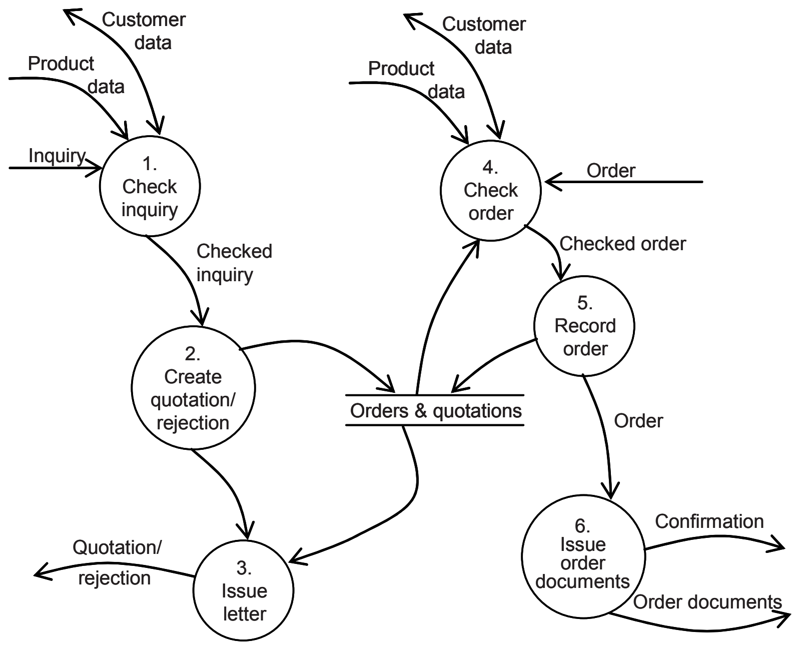
Figure 5-14 **SA context diagram**



Level-1 and level-2 diagrams

In the next steps, the top-level processes are refined into more detailed processes. Figure 5-16 shows the refinement of the "check inquiry" process (no. 1) into three subprocesses numbered 1.1 to 1.3. The other level-0 processes will be refined in the same way. Afterwards, level-1 processes will be refined, and so on. How many levels of refinement are appropriate depends on the complexity of the problem. A rule of thumb says refinement should stop when a process can be described in less than one page of *structured English*. (Structured English is a rigorously defined subset of natural English with a restricted vocabulary and conjunctions restricting control flow to structured-programming like constructs.)

Figure 5-15 **SA level-0 diagram**



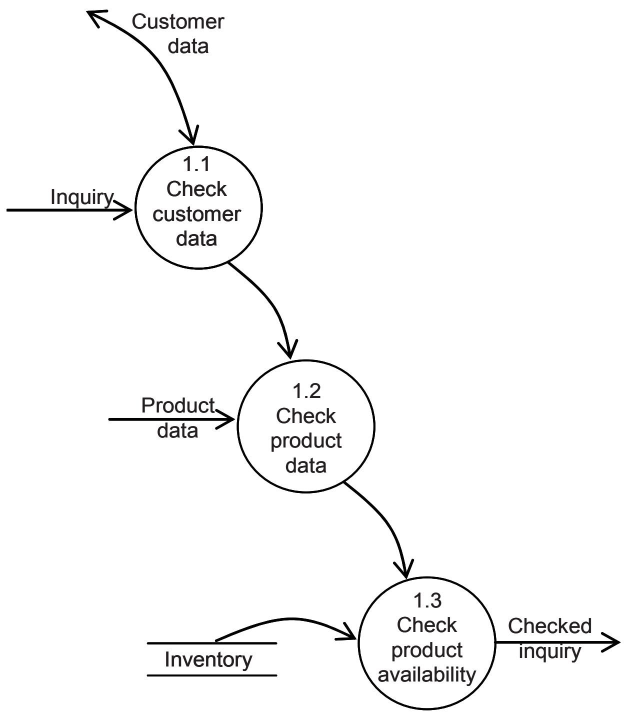
DFDs are the main modeling technique in SA

DFDs are the essential modeling technique in SA. A data dictionary, structured English, decision trees and decision tables are additional techniques to supplement the DFDs with more detailed information.

CASE tools supporting structured analysis usually provide graphical support for creating the respective documents.

The brief description of structured analysis in the above paragraphs was intended to show SA's basic approach of refining processes with the help of data flow diagrams. Readers requiring more indepth information on structured analysis are advised to consult dedicated SA books [e.g. DeMarco 1978, Gane 1979, Yourdon 1989] or ISD books with a structured-analysis focus [e.g. Kendall 2005].

Figure 5-16 **SA level-1 diagram decomposing process no. 1**



Business area analysis (BAA)

Information engineering (IE)

Business area analysis (BAA) is a part of *information engineering (IE)*. IE is a comprehensive model-based approach to enterprise-wide planning, analysis, design and construction of information systems, proposed and marketed by James Martin since the early 1990s. As described in section 4.2.3, information engineering has four major stages:

– Information strategy planning
– Business area analysis
– System design
– Construction

On the top level of the pyramid (cf. figure 4-7), in *information strategy planning (ISP),* models representing the strategic opportunities, goals, critical success factors and information needs of different parts of the enterprise are developed. The whole enterprise is modeled in this way on a high abstraction level and then split into different areas appropriate for business area analysis.

Business area: "a naturally cohesive grouping of business functions and data"

Since IE is essentially a data and function-oriented approach, a business area is defined as "a naturally cohesive grouping of business functions and data" [Martin 1990, p. 184]. What makes up a business area can be determined, for example, by a clustering algorithm. A business area is not an organizational unit of the enterprise but something defined in terms of interconnected functions and data, possibly spanning across several departments or organizational units. A business area such as warehouse management, for example, will touch the warehousing, purchasing, production-planning and quality-control departments.

Business area analysis assesses individually each part of an enterprise, i.e. each business area is analyzed one after the other. Priorities, i.e. which business area to start with, are determined within information strategy planning. The purpose of BAA is to develop a number of models documenting the results of the analysis. The main model types are:

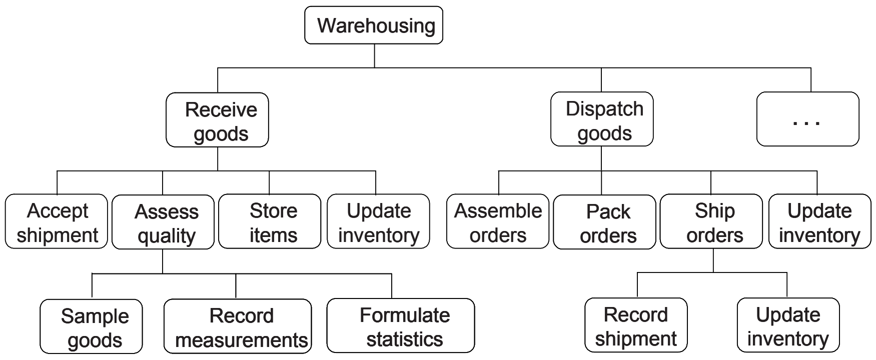
Data model

- *Data model* – essentially a normalized entity-relationship model in a particular notation, using attribute-free relationships and crow's-foot connectors for cardinalities [Martin 1990, pp. 304-317]. (ER models are briefly discussed in the section on ARIS below.)

Process decomposition model

- *Process decomposition model* – a static model of the top-level business functions decomposed into lower-level processes by functional decomposition, creating a tree-structured hierarchy of processes. (In IE terminology, the top-level activities identified in ISP are called business functions. A process is an activity that transforms input data into output data as in SA.) An example of a decomposition diagram is given in figure 5-17. In this diagram, the business function "warehousing" is refined in three levels of processes.

Figure 5-17 **Process decomposition diagram (example)[§]**

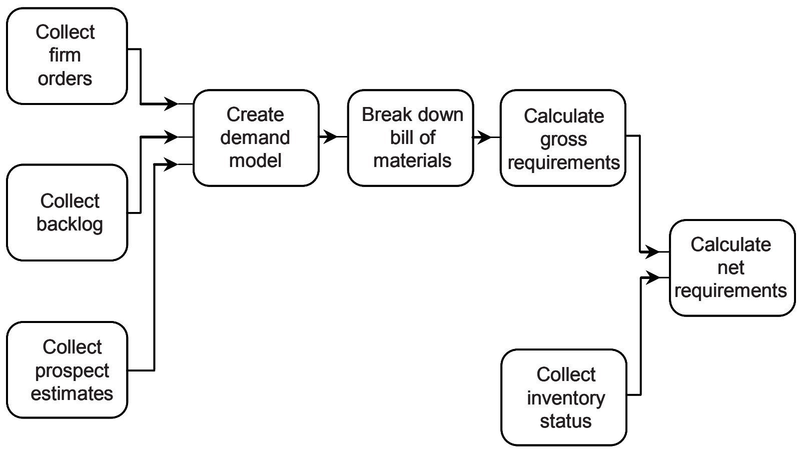


Process dependency model

- *Process dependency model* – a model that shows the dynamic relationships between processes. Some processes depend on others in the sense that they can be executed only if some other process has been performed before. An example from material requirements planning (MRP) is shown in figure 5-18. Process dependency models can be refined and specified very precisely with the help of cardinalities and logical connectors.
- *Process data flow model* – a process dependency model with data inputs and outputs added to the processes. Figure 5-19 shows an example of a process data flow diagram. It was created from a process dependency diagram with data flows added. (The bold circles mean mutual exclusivity. After checking product availability, either the "create backorder" process or the "fill order" process has to be performed.) Note that a process data flow diagram is not the same as a data flow diagram in structured analysis because the sequence of processes is not defined by data flows.

§ Martin 1990, p. 259.

Figure 5-18 **Process dependency diagram (example)**[§]


Collect firm orders
Collect backlog
Collect prospect estimates
Create demand model
Break down bill of materials
Calculate gross requirements
Calculate net requirements
Collect inventory status

Process/data matrix (process/entity matrix)

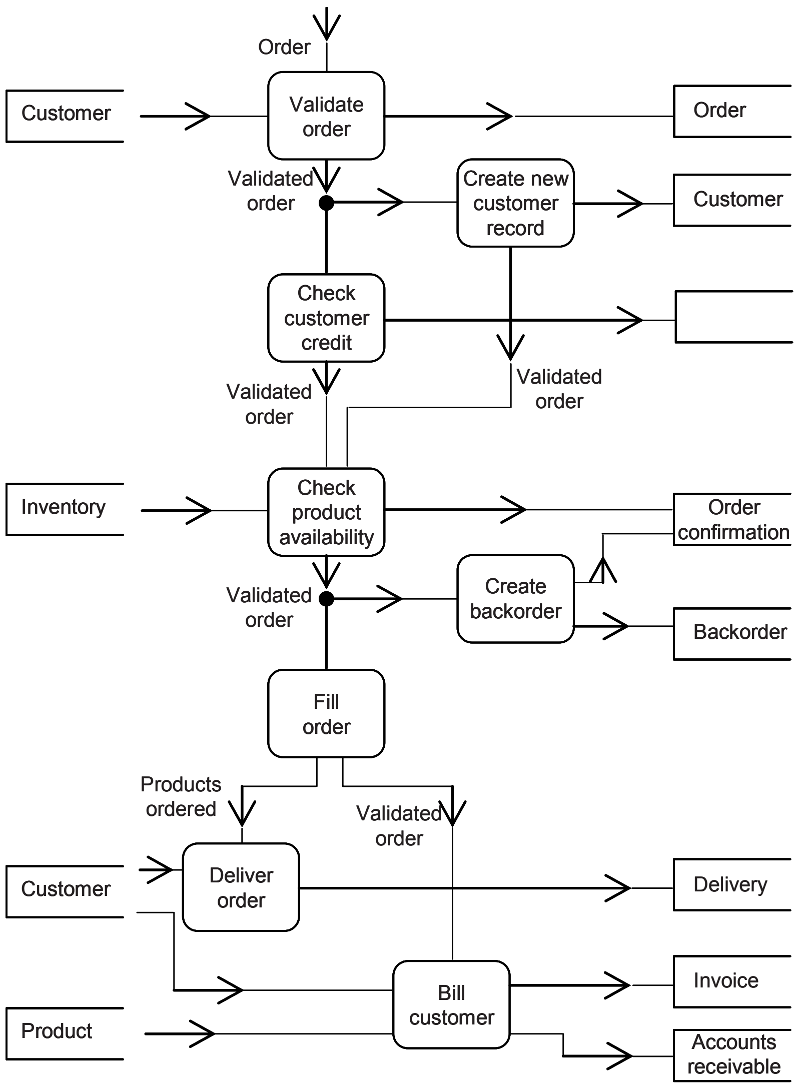
- *Process/data matrix (process/entity matrix)* – a matrix showing which processes create (C), read (R), update (U) or delete (D) which data entities. The matrix rows signify processes and columns signify entity types. Entries in the cells are the above abbreviations C, R, U and D (sometimes referred to as CRUD). Some cells have one letter, some more. Figure 5-20 shows a portion of a process/data matrix associated with figure 5-19[#]. For example, the "Customer" column has obviously a "C" entry in the "Create new customer record" row and an "R" for the delivery and billing processes.

The models of BAA are not created in a linear sequence but in iterations with successive refinement. Completeness and consistency of the models is checked in this process. Computerized tools (CASE tools) do some of the checking automatically.

§ Martin 1990, p. 266.

# Some data flows coming from or going to data stores were shown explicitly in figure 5-19. In addition, we assume that orders are kept in a data store between processes. Each process that needs to do something with an order will read the order from the data store (R) and possibly update the order and write it back to the data store (U).

Figure 5-19 Process data flow diagram (example) [Martin 1990, p. 271]



Completeness and consistency checks based on the encyclopedia

This is possible since all models and all their components are (ideally) stored in a central repository, the encyclopedia that was illustrated in figure 4-8. All models are interconnected via the encyclopedia.

Figure 5-20 **A section of a process/data matrix**

Process \ Entity Type	Order	Customer	Inventory	Product	. . .
Validate order	CU	R		R	
Create new customer record		C			
Check customer credit	RU				
Check product availability	RU		R		
Create backorder					
Fill order	RU		U		
Deliver order	RU	R			
Bill customer	RD	R		R	
. . .					

Legend:

C = Create
R = Read
U = Update
D = Delete

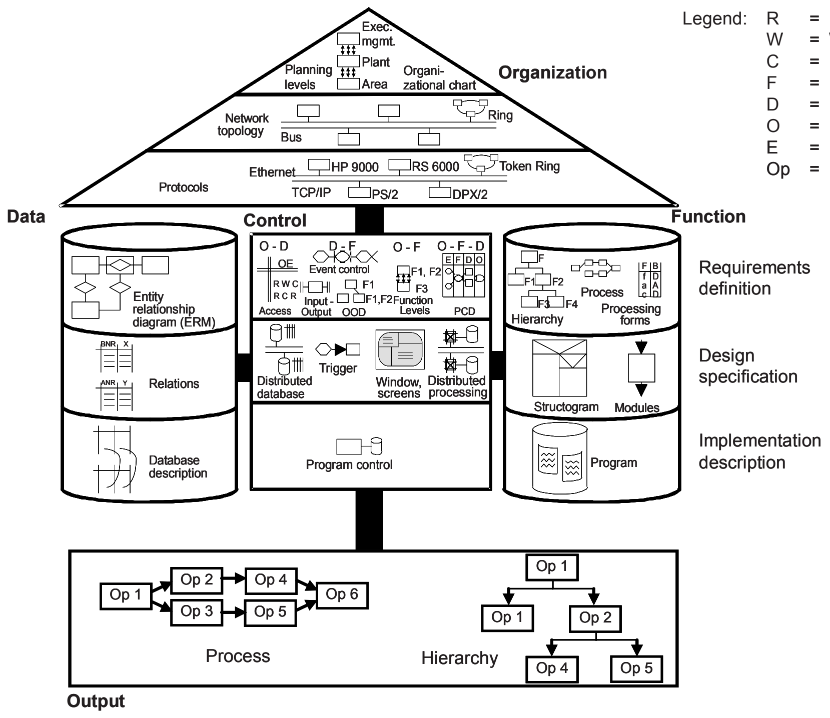
In this way it can be checked whether an entity referred to in a process data flow diagram has been created somewhere, i.e. it must have a "C" in some process/entity matrix. The attributes of an entity type specified in an entity-relationship diagram are available in the process data flow diagram. A process named in the matrix must be specified in a decomposition or process dependency diagram.

ARIS requirements definition

ARIS (architecture of integrated information systems) is similar to information engineering, capturing different views of an organization's information systems. It starts by identifying business processes and modeling them. ARIS uses a comprehensive set of methods. These methods are often summarized in a picture shown in figure 5-21, called the ARIS HOBE (house of business engineering) [Scheer 2002, p. 4].

Event-controlled process chains (EPCs)

Figure 5-21 **ARIS HOBE (house of business engineering)[§]**



Legend:		
	R	= Read
	W	= Write
	C	= Create
	F	= Function
	D	= Data
	O	= Organization
	E	= Event
	Op	= Output

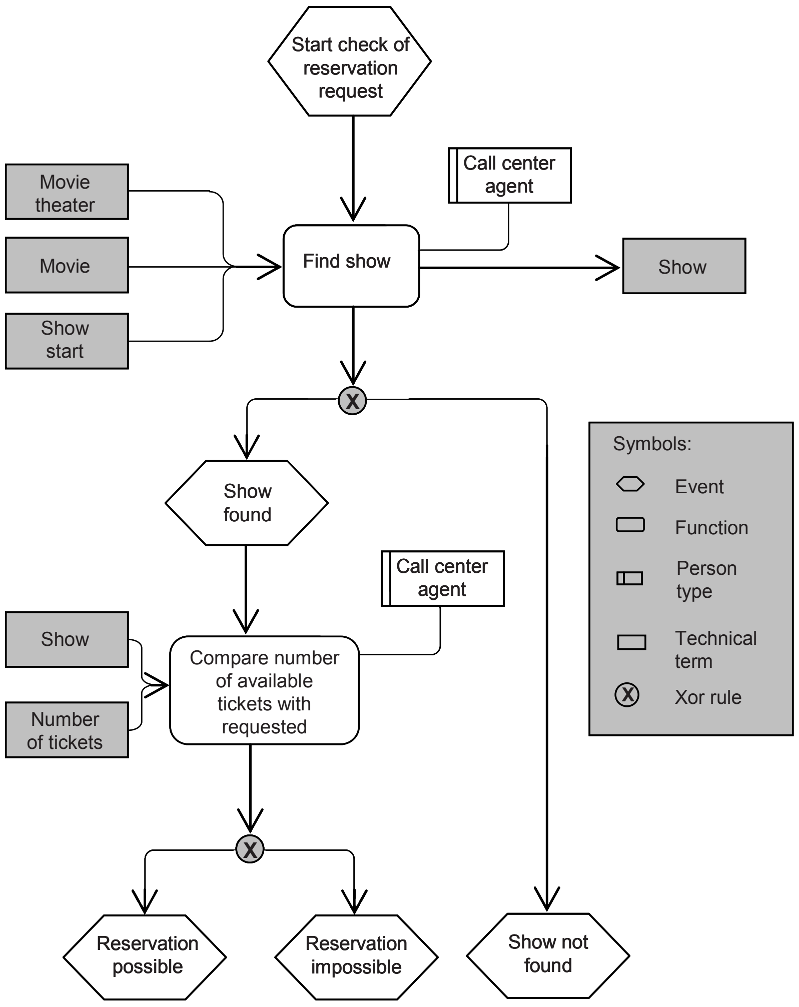
Event-controlled process chains (EPCs) serve as the major initial modeling technique. In figure 5-22, a small example of an EPC is shown,

Event-controlled process chains

§ Scheer 2002, p. 4.

illustrating its main elements: functions, events, person types (or organizational units) and technical terms (inputs/outputs of EPC functions).

Figure 5-21 **Event-controlled process chain (example)**[§]



§ Andres 2006, p. 4.

ARIS views and descriptive levels

Just like information engineering, ARIS is essentially data and function-oriented, starting from a business perspective. The ARIS approach is to identify functions, data and organizational units involved in the business processes and their interconnections, and to represent them in five *views*, the function, data, organization, control and output views. Each view except the last one is examined on three different abstraction levels (called descriptive levels). These *levels* are requirements definition, design specification and implementation description.

EPCs represent the business view of an organization's processes. They can be interpreted as high-level descriptions of the requirements for information systems development. In order to make operational requirements out of the high-level process descriptions, two approaches are available: 1) the original ARIS approach and, 2) the UML oriented approach.

1) Original ARIS approach

ARIS models for requirements engineering

For each view and each level of ARIS, models and methods for describing the respective elements are provided (cf. figure 5-21). For requirements engineering, the respective models and methods are provided on the ARIS "requirements definition" level, including the following:

- Entity-relationship models (for the data view)
- Decomposition diagrams and process-sequence diagrams (for the functions view)
- Organizational charts (for the organization view)
- Process data flow diagrams, EPCs, function/data matrices and other matrices (for the control view)

Most of these models and diagrams are neither new nor ARIS specific. For example, decomposition diagrams, organizational charts, process data flow diagrams and entity-relationship diagrams are used in information engineering as well. ARIS unites them into a comprehensive framework in a similar way to IE.

Entity-relationship model (ERM)

Since the entity-relationship model (ERM) has been the most widely used data-modeling technique for many years, we will explain it here briefly. (More detailed descriptions can be found in the literature [e.g. Elmasri 2006, Hoffer 2006, Bagui 2003]). The original entity-relationship model was developed by P.P. Chen in the 1970s [Chen 1976]. A number of variants have come into existence since then. One of them is the variant used in information engineering as mentioned above.

ERMs are quite similar to domain models in UML. This is not surprising, since the entity-relationship model has long been employed for the same purpose as a UML domain model. The basic concepts are entity types, relationship types and attribute types; or on an instance level: entities, relationships and attributes.

Entities and entity types

Entities are real or abstract objects which are of interest in the domain which is to be modeled. For example, the book "The making of information systems", the author "Karl Kurbel", or the specific book shop at the corner are all entities. Entity types are types of such objects, e.g. the types "book", "author" and "book shop".

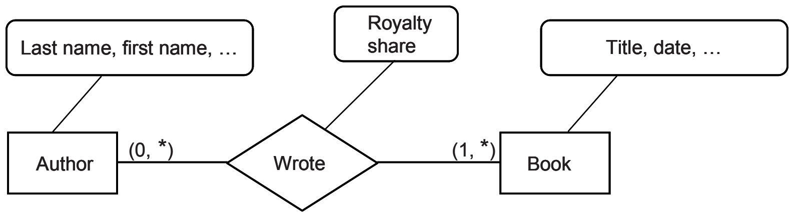
Relationships and relationship types

Relationships are logical connections between entities (for example, "Karl Kurbel wrote The making of information systems"). Relationship types are the types of such connections between entity types. For example, the relationship type "wrote" will connect the entity types "book" and "author" as shown in figure 5-23.

Attributes and attribute types

Attributes are properties of an entity or a relationship. Attribute types are the types of these properties. For example, "last name" is an attribute type of the entity type "author". Note that in the ER model version which is supported by ARIS, both entities and attributes can have attributes. In the example of figure 5-23, the "wrote" relationship has an attribute "royalty share" which specifies the percentage from the total royalty a (co-) author will receive regarding this particular book.

Figure 5-23 ERM for books and authors



Min-max cardinalities

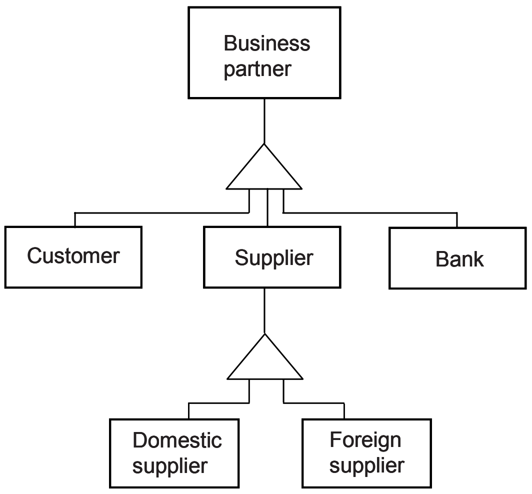
Relationship types have *cardinalities* (or *complexities*). A cardinality is similar to a multiplicity in a UML class diagram, defining with how many entities of type 2 a particular entity of type 1 may be connected. A precise way of writing cardinalities is the *min-max notation* (min-max cardinalities) specifying both the minimum and the maximum number of connected entities. Min-max cardinalities are noted down as tuples (min, max).

In figure 5-23, the (0,*) cardinality on the left says that one particular author can have written zero or more books. (1,*) on the right means that one particular book must have at least one author and can have many authors. It is a matter of convention where on the connecting lines the cardinalities are written (next to entity type 1 or entity type 2). We follow Scheer's usage here (i.e. writing the (0,*) cardinality on the author's side), but others prefer to do it the other way round (i.e. writing the (0,*) cardinality on the book's side). Note that Scheer's convention is opposite to the way in which the corresponding multiplicities were written in the class diagram of figure 5-11.

"Is a" relationship

In addition to the basic concepts just described, common extensions are generalization/specialization ("is a" relationships) and aggregation (re-interpreted relationship types).

Figure 5-24 Generalization/specialization (example)



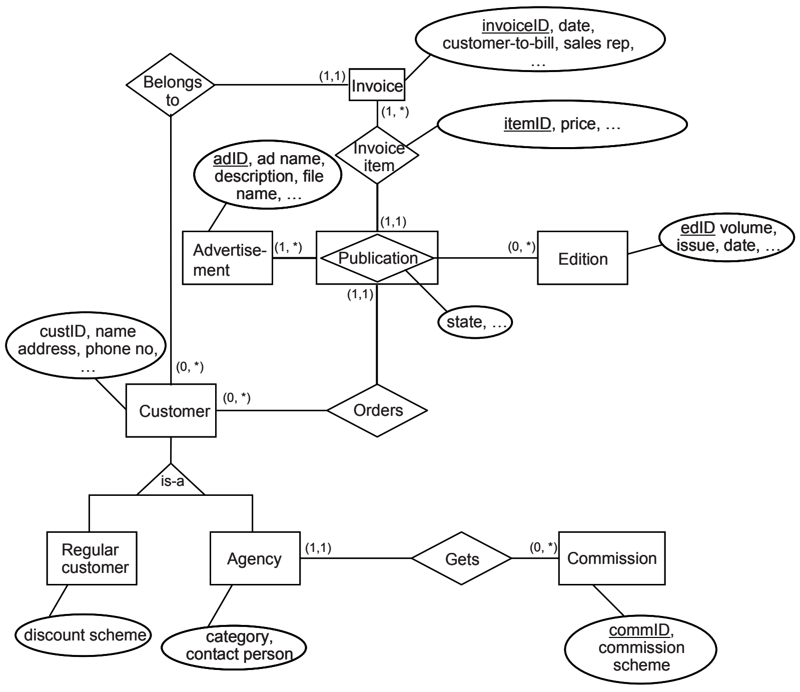
Aggregation

Aggregation is a concept often used with different meanings. In the ARIS entity-relationship model, it stands for creating new entity types from relationships, i.e. related entity types are combined into an aggregated type. The purpose of this combination is to have a new concept (a newly constructed entity type) that itself may have relationships with other entity types.

Re-interpreted relationship types

An example is given in figure 5-25. The problem underlying this entity-relationship diagram is managing advertisements in a newspaper company. The same advertisement can be published several times in different editions of the newspaper. "Publication" is primarily a relationship type connecting advertisements with newspaper editions. Since customers can place orders for the publication of an advertisement in one or more editions of the paper, obviously a relationship between the entity type "Customer" and the relationship type "Publication" is needed. However, relationships exist only between entity types. Therefore, the relationship type "Publication" has to be re-interpreted as an entity type, which is indicated by a (entity) rectangle around the (relationship) diamond.

Figure 5-25 ERM with aggregation and generalization



Likewise, a relationship between a particular publication of an advertisement and an invoice for it can only be established if the publication is indeed an entity type. For this purpose the re-interpretation of "Publication" as an entity type is also needed.

Attributes identifying an individual entity uniquely

Some attributes in the figure are underlined. These attributes are used to identify an individual entity uniquely. In the real world, all objects of a business have such identifiers. For example, a particular invoice is uniquely identified by an invoice number. In the ERM, the respective attribute is called "invoiceID".

2) UML-oriented approach

Transformation of EPCs into use cases and class models

Many organizations today use UML for all modeling and specification tasks in information systems development. Taking this fact into account, ARIS supports the transformation of EPCs defining business processes into use cases and UML models [Andres 2006, pp. 5-6]. Essentially the creation of a use-case model and a class model from the event-controlled process chains is supported.

The use-case model is created through a mapping of EPC functions onto UML use cases in a schematic way. The functions are treated as if they were use cases and copied to a use-case diagram. The class diagram is created from the technical terms of the EPC, which represent data. The technical terms are considered candidates for classes. They are examined and then made either classes or attributes in the class model.

5.2 Design

Design leads to a description of the software

The goal of design is to draft a solution in terms of software concepts. While requirements engineering captures and specifies the requirements for the systems as seen by the stakeholders, *design* activities ultimately lead to a description of the future *software system*. Elements of the solution are software concepts, not user or domain concepts as in requirements engineering. Nevertheless, there is a correspondence between domain concepts and software concepts, as the latter ones are usually derived from the former ones. The result of the design is a specification of the software system that can be directly transformed into executable code in the implementation stage.

Boundaries between requirements and design are blurry

The term "design" has blurry boundaries, obscuring where requirements end and design specifications begin. Even in a well-known approach such as RUP (Rational unified process; cf. section 4.3.2), the analysis and design disciplines are not cleanly separated. Some of the UML modeling techniques for analysis and design are the same.

How do requirements turn into design specifications?

How do requirements turn into design specifications? How do layers, modules, classes, objects or whatever constructs are needed come into existence? In the majority of the software engineering literature, there used to be a gap between requirements engineering and a requirements specification on the one hand, and follow-up activities in design on the other hand. Much was left to the creativity, intuition and experience of the designer. Operational approaches were available from authors from the SA/SD (structured analysis/structured design) field, but not really accepted by hard-core software engineering gurus. In section 5.2.5 we will discuss these approaches briefly.

Patterns

Eventually, the intuition and experience-based approaches by expert designers found their explicit descriptions in so-called *patterns*. A pattern is an effective solution to a problem that occurs repeatedly in a given context. It is accompanied by a description of what the consequences of using the pattern are and by a set of "known uses" [Kircher 2007, p. 29].

In short, a pattern is a general repeatable solution to a commonly occurring problem. It describes a category of problems and is meant to be reused in new contexts. A definition of the term is as follows:

Definition: pattern

> A *pattern* is a named description of a problem and a solution that can be applied in new contexts, with advice on how to apply it in novel situations, taking into consideration the forces and tradeoffs in varying circumstances [Larman 2005, p. 279].

Patterns became popular in the 1990s. The best-known set of patterns is the so-called GoF patterns collections discussed further below. Patterns serve different purposes on different abstraction levels. Rough categories are [Buschmann 1996, pp. 11-16]:

Pattern categories

» Architectural patterns – applied in early design activities in order to establish an architecture
» Design patterns – related to the design of partial solutions of a certain type

» Idioms – used for low-level design solutions

Since the patterns discussion began in the 1990s, thousands of patterns have been created and used by software developers. Heavy books have been edited, such as further volumes of the cited "Pattern-Oriented Software Architecture, Volume I" [Buschmann 1996], and a series of PLoP (pattern language of programming) conferences has been held since then.

In software development projects, many patterns are used together, often in repeated combinations. This has lead to a distinction between stand-alone patterns and pattern collections [Buschmann 2007, p. 32].

Pattern collections

Pattern collections contain several patterns which are related with each other in a specific relationship:

» Pattern compounds – collections of pattern which are usually applied together.
» Pattern sequences – collections of patterns that are always used in the same sequence.
» Pattern complements – collections where one pattern provides either the missing ingredient for another pattern or a balanced solution for a related problem (e.g. a pattern for creating and a pattern for destroying an object).

Pattern languages

Since the relationships between patterns can be quite complex (e.g. complements consisting of compounds and sequences), so-called *pattern languages* have been created. Pattern languages aim to provide holistic support for using patterns to develop software for specific domains, such as e-commerce and communication middleware [Buschmann 2007, p. 33].

More patterns will be created

In the pattern community, it is expected that more patterns will come into existence, focusing on particular application domains (e.g. business process modeling, transaction-oriented business systems, mobile systems) and software technologies (e.g. programming languages). Many software developers use patterns today, but even more do not [Manolescu 2007, p. 62]. However, the dissemination of pattern-oriented approaches in practice is expected to grow.

In the following description, we will focus on those perspectives of design activities that are of primary interest in the development of business information systems: designing the system architecture, the classes and objects, the user interface and the database. Patterns play an important role in these perspectives, but other approaches will be discussed as well.

5.2.1 Architectural Design

Architecture-in-the-large

The importance of an appropriate architecture has already been emphasized in chapter 3. In this section, we will discuss briefly how an architecture is developed. Actually, an *architecture-in-the-large* may already be prescribed for the current development effort, i.e. when the information system under discussion has to match existing systems and fit into a given information systems landscape. For example, if the company has already implemented an organization-wide enterprise service-oriented architecture (ESOA, cf. section 3.4), then the new system will have to match this architecture.

Architecture-in-the-small

However, architectural decisions will still be needed for the current information system. At least an *architecture-in-the-small* has to be developed unless the system is trivial. The question when this architecture is created has several answers. A high-level architecture may be created in early project stages and later refined. In the RUP process model, for example, a high-level architecture is established in the inception phase. Later in the elaboration phase, this architecture is stabilized and an architectural prototype is built (cf. section 4.3.1). Distinguishing between a logical architecture and a physical architecture is helpful to identify the necessary tasks.

Logical architecture

A *logical architecture* deals with the functionality of the system, allocating functionality to different parts of the system [Eriksson 2004, p. 254]. It reflects the application logic, but not the physical distribution of that logic into software components. The logical architecture mainly specifies the functional properties of the system and is driven by the *functional requirements* (cf. section 5.1.1).

Several UML diagram types are used together to describe a logical architecture, including package, use-case, class and sequence diagrams. Activity, state machine and communication diagrams can also be employed. (The latter ones are explained further below in the context of design tasks.)

Package diagram

Package diagrams group together architectural components which belong together. A package in UML is a general grouping mechanism for all kinds of model elements. The package content (i.e. all package elements belonging to the package) is drawn inside a large box with a small rectangle (a tab) attached to its upper left corner. The tab may contain the name of the package. Package elements can be packages themselves. This is usually the case when a logical architecture is displayed in a package diagram.

Figure 5-26 shows an example of a logical architecture consisting of three layers. Each layer contains further packages. Since packages are general grouping mechanisms, the contained packages may consist of any type of model elements. A reasonable assumption in the case of figure 5-26 would be that the contained packages contain classes derived from the conceptual classes of the domain model. The dashed lines represent dependency relationships.

Physical architecture

A *physical architecture* describes the physical structure of the software system, i.e. the software components implementing the functional concepts specified in the logical architecture and the relationships among the components.

In a broader sense, the physical architecture includes hardware, network and system software related aspects such as the distribution of the run-time software in terms of processes, programs and network nodes. The physical architecture mainly deals with the *non-functional properties* of the system such as reliability, compatibility, resource usage and deployment of the system.

The physical architecture is created from the logical architecture by mapping the models describing the logical architecture onto models for the physical architecture. Classes and other concepts specified in the logical architecture are connected with the respective artifacts in the physical architecture.

UML diagrams for a physical architecture

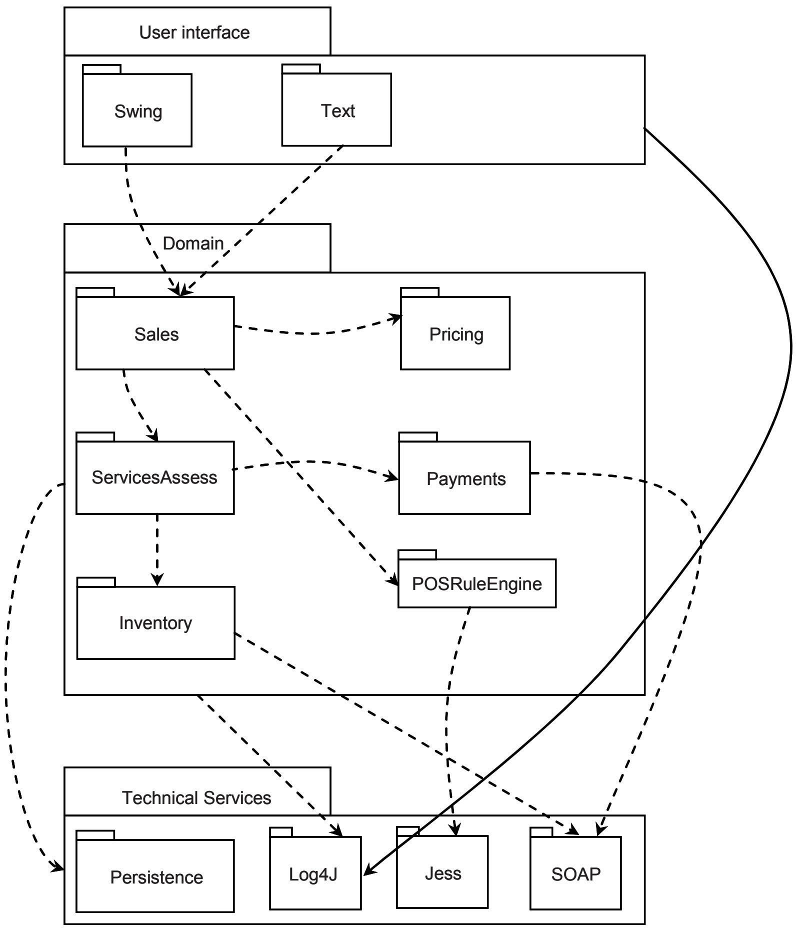
UML diagrams used to illustrate a physical architecture are the same as those used for a logical architecture: package, use-case, class, interaction, state machine and activity diagrams. Interaction diagrams, in particular communication diagrams, help to make the points visible where layer (package) boundaries are crossed. For this purpose, architecturally significant scenarios should be developed [Larman 2005, p. 564]. Information needed for the implementation is provided through component and deployment diagrams.

RUP: architectural analysis (AA)

Creating an architecture has been recognized as a process of its own, in addition to or as part of a general process model. In RUP, an activity called *architectural analysis (AA)* addresses the architecture issues. The purpose of AA is:

1. to identify factors which influence the architecture,
2. to make decisions resolving the issues.

Figure 5-26 **Package diagram showing a three-layer architecture**§



Factors with architectural implications identified in the first step are called *architectural factors* or architecturally significant requirements. The result of an architectural analysis, essentially a documentation of

§ Larman 2005, p. 563.

the architectural decisions, is called a *software architecture document (SAD)*.

Bass et al.: attribute-driven design (ADD)

Another approach to designing an architecture was proposed by Bass, Clements and Kazmann under the name *attribute-driven design (ADD)* [Bass 2003, pp. 155-166]. ADD starts from the so-called *architectural drivers*. In the terms of AA, these are architecturally significant requirements that have a dominating influence on the design of the architecture. For example, high-availability requirements can be an architectural driver for a point-of-sale system. ADD is basically a method that decomposes a system top-down into subsystems or modules, looking at one module at a time and refining it.

Architectural patterns

Architectural patterns are useful aids in developing an architecture. An architectural pattern describes a fundamental structural organization or schema for software systems. Buschmann et al. in their seminal book "Pattern-oriented software architecture" define an architectural pattern as "... providing a set of predefined subsystems, specifying their responsibilities, and including rules and guidelines for organizing the relationships between them" [Buschmann 1996, p. 26].

Architectural patterns [Buschmann 1996]

Architectural patterns have been proposed by many authors and for many purposes. The most cited classification of patterns is the one by Buschmann et al., covering a variety of application areas [Buschmann 1996, pp. 25-220]. These authors provide a detailed discussion of architectural patterns. Short summaries and abstracts of their patterns can be found in many guidelines about architectural patterns, such as the subsequent ones, taken from the Hillside website [e.g. Hillside 2007]:

- *Layers* – this pattern helps to structure applications that can be decomposed into groups of subtasks, where each group of subtasks is at a particular level of abstraction.
- *Pipes and filters* – provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data are passed through pipes between adjacent filters. Recombining filters allows the developer to build families of related systems.
- *Blackboard* – useful for problems for which no deterministic solution strategies are known. In a blackboard architecture, several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- *Broker* – used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication,

such as forwarding requests, as well as for transmitting results and exceptions.

- *Model-view-controller (MVC)* – dividing an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.
- *Presentation-abstraction-control (PAC)* – defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.
- *Microkernel* – this pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.
- *Reflection* – provides a mechanism for changing the structure and behavior of software systems dynamically. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

While some of these pattern refer to specific tasks, e.g. the pipes-and-filters pattern for systems that process data streams, others describe typical problems and solutions for business information systems. Common patterns are the layers, broker and model-view-controller patterns. *Layers* (or tiers) as addressed in the layers pattern, which divide a system into levels of abstraction, were discussed in section 3.2. A well-known *broker* architecture is CORBA (common object request broker architecture) [OMG 2007].

Model-view-controller (MVC)

The *model-view-controller (MVC)* pattern is a very common pattern for interactive systems focused on presenting data to the user, for example systems with a web front-end, which by itself is a comprehensive task. Since many application systems need to support multiple types of users with multiple types of interfaces, the "normal" web front-end is only one interface. An electronic shop, for example, may require an HTML solution for web customers, a WML (wireless markup language) front-end for wireless customers, a Java Swing interface for system administrators, and an XML-based web service for suppliers of the shop [Sun 2002]. No matter which channel an interaction goes through (e.g. an HTTP request, a SOAP message), all will need the same core functionality of the shop system.

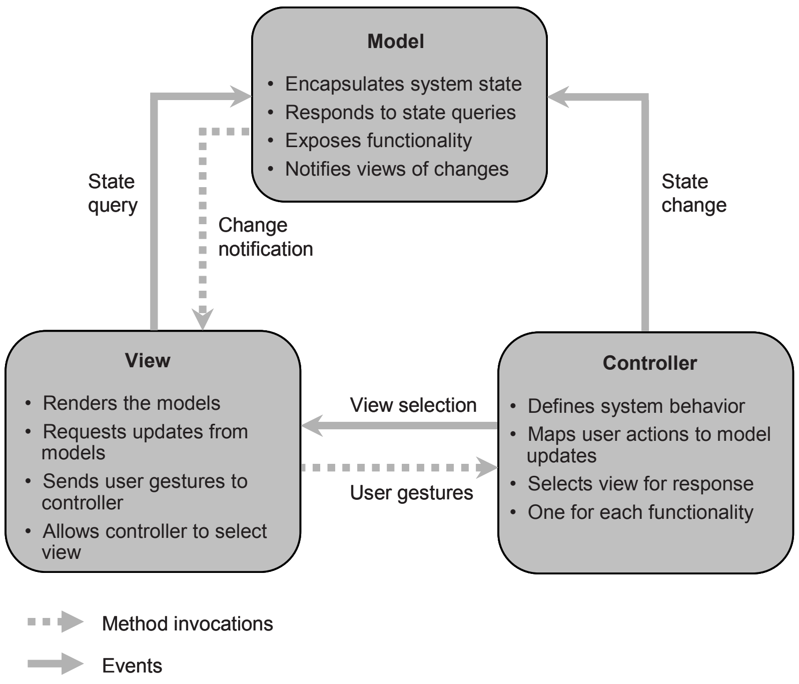
Instead of repeating the functionality and the database in respective HTML, WML, XML and Swing-based solutions, the MVC pattern divides the system into three main components. The rationale is to separate data concerns from user interface concerns, so that changes to the user interface do not affect the data handling. Vice versa, the data can be reorganized without changing the user interface.

The MVC pattern maps traditional application tasks – input, processing and output – to the graphical user interaction model. It decouples data access and business logic from data presentation and user interaction with the help of an intermediate component: the controller. As shown in figure 5-27, the three components of an MVC architecture are [Singh 2002, p. 348, Gulzar 2002, p. 2]:

- *Model* – contains a domain-specific representation of the information on which the system operates and the core functionality of the system. The model represents enterprise data and the business rules that govern access to and updates of this data. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model. Model
- *Views* – one or more displaying information to the user. A view renders the contents of the model. It accesses the database through the model and specifies how the data should be presented. The view is responsible for maintaining consistency in its presentation when the model changes. Views
- *Controllers* – one or more defining the system behavior, responding to events, especially user actions. A controller translates interactions with the view into actions to be performed by the model. The controller dispatches user requests and selects views for presentation. It interprets user inputs (e.g. HTTP get and post requests) and maps them onto actions to be performed by the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view. A system may have more than one controller. Some systems use a separate controller for each Controllers

client type, because view interaction and selection often vary between client types.

Figure 5-27 **Model-view-controller architecture [Singh 2002, p. 348]**


Model
• Encapsulates system state
• Responds to state queries
• Exposes functionality
• Notifies views of changes
State query
Change notification
State change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View selection
User gestures
Controller
• Defines system behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method invocations
Events

Java pet store (Sun)

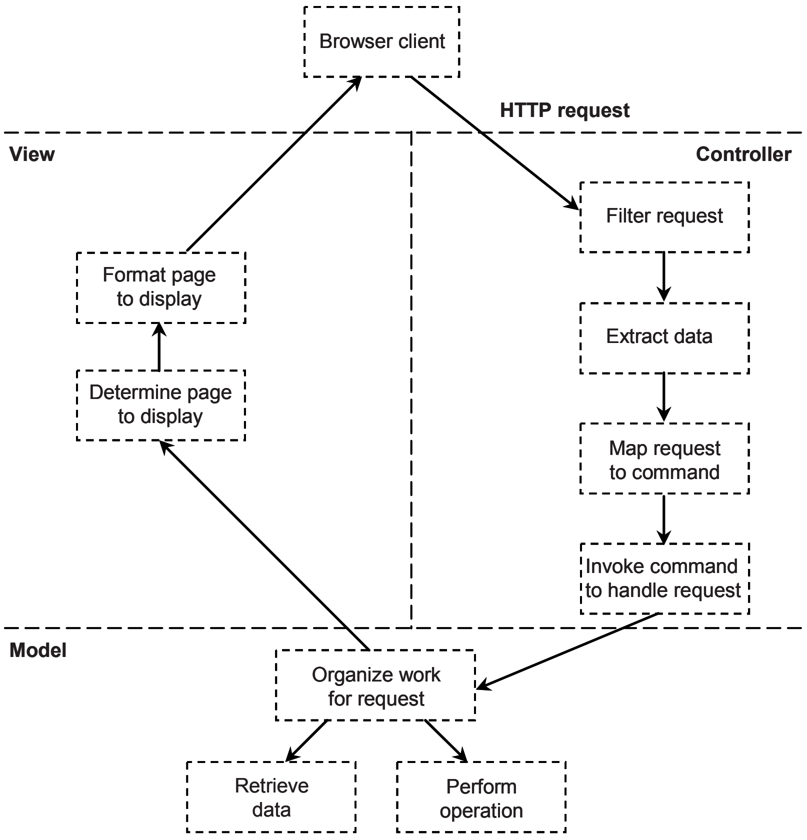
An often used example to demonstrate the model-view-controller architecture is the *Java pet store*. This is a sample system provided by Sun Microsystems. It has been used to demonstrate various aspects of the Java platforms over the years. (The latest version of the store illustrates, for example, how to develop an AJAX enabled Web 2.0 system.)

The Java pet store is a typical e-commerce system – an online store based on an electronic product catalog organized by categories. Users browsing the catalog get various views of products and services for sale. The store takes orders, acknowledges orders and processes credit-card payments. For these tasks, it has to manage user logins, shopping sessions, personalization and shipping information. The Java pet store also includes administration functions such as inventory and order management.

An MVC architecture was chosen for the pet store because it provides a structure for handling complex, presentation-oriented problems. The pet store's website represents such a problem. It has numerous views and pages available for customer access, potentially written in different languages, plus content that may be personalized.

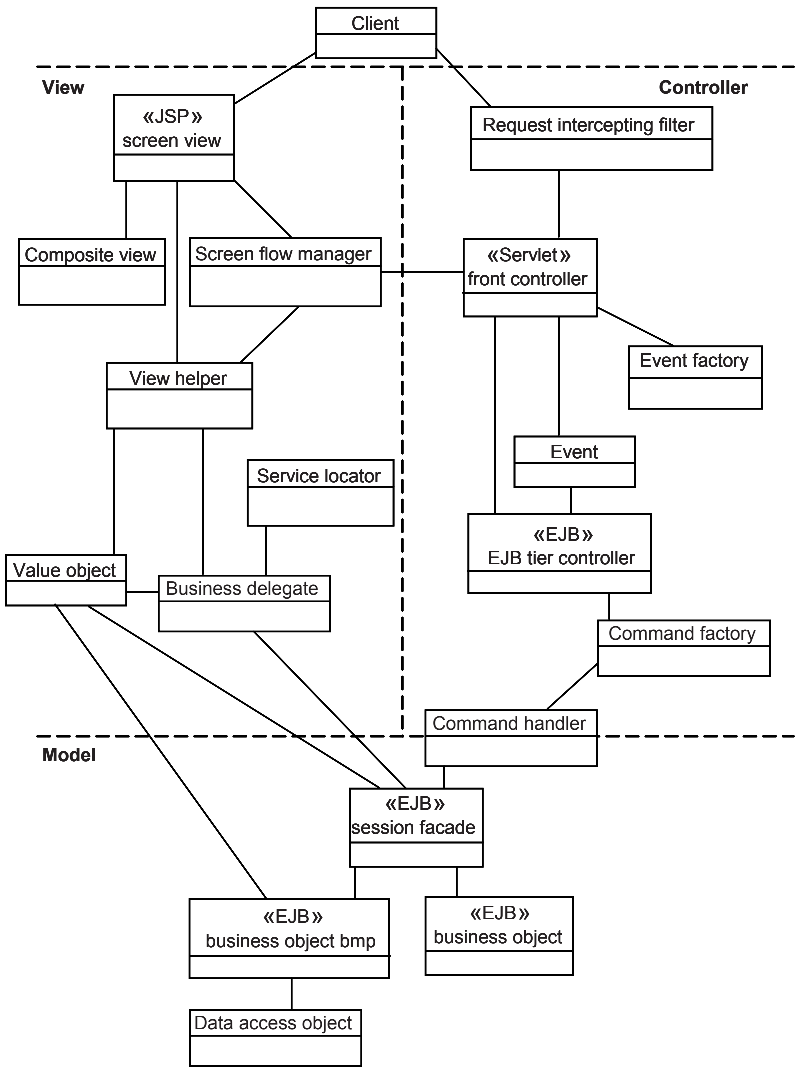
MVC facilitates the handling of complex, presentation-oriented problems

Figure 5-28 **Model, view and controller tasks in Java pet store**[§]



§ Singh 2002, p. 366.

Figure 5-29 **Class diagram showing MVC classes in Java pet store**§



§ Adapted from Singh 2002, p. 366.

Customers can make many different requests, each of which the store interprets and carries out, with data coming from multiple sources. The system dynamically determines the sequence of views to display to the customer.

Controller, model and view-related tasks

The flow of tasks related with these features is roughly divided into three sections: controller, model and view-related tasks as shown in figure 5-28. HTTP requests coming from the user's browser are examined by the controller, leading to an invocation of a suitable command. The model is responsible for performing the respective operation and retrieving the data needed for the operation. View-oriented tasks are determining which page to display and formating the page that is finally shown to the user as the result of the initial request.

Figure 5-28 is a flow-oriented description and not an architectural specification of classes and relationships between the classes. It serves as a first approximation of which tasks are there to be solved and to which part of the solution these tasks belong. The latter assignments are not always unambiguous. For example, determining the page to display next will involve not only the view but also the controller.

From the rough division of tasks into the three parts, an architecture of the pet store can be developed and described in appropriate UML models and diagram. A class diagram, for example, will be created, specifying the major architectural components in the form of classes and relationships. Figure 5-29 shows a high-level class diagram indicating which classes belong to which part (model, view, controller). We will not discuss the details of the architecture here. Since the pet store is a reference system for Java developers, the solution contains advanced features such as Java EE design patterns beyond the scope of this book.

The reader interested in more information will find details of the architecture in the book by Singh et al. [Singh 2002, chapter 11] and details of the current implementation of the Java pet store on Sun Microsystems' developers website [Sun 2006a].

5.2.2 Designing the Objects (Design Model)

Use-case realization

The main goal of design is to specify a software solution capable of performing the use cases. In RUP, the term *use-case realization* is used for this purpose. A use-case realization describes how a particular use case is realized within the design model. Following the object-oriented paradigm, design means in the first place that the classes of the software system and the interaction between objects of these classes need to be specified. In fact, the core of "traditional" definitions of object-oriented design (i.e. before RUP and UML) was to find an appropriate class and object structure [e.g. Booch 1994, p. 39].

Design classes

Classes and their relationships are modeled in class models and represented by *class diagrams*. These diagrams were already introduced in section 5.2.1 above where they served to describe domain concepts. The same notation is used in design, but here the classes are *design classes*, which means that they are intended to be implemented later in a programming language. Design classes have more detail than domain concepts because the goal of design is to create complete, consistent and unambiguous specifications.

Interaction diagrams

Interaction diagrams (sequence and communication diagrams) define how objects are supposed to interact by sending, receiving and processing messages in order to fulfill the requirements. Sequence diagrams were introduced as a diagramming technique for requirements engineering in section 5.1.4. In design, the same type of diagram is used but contains more detail. Communication diagrams are discussed further below.

Design model: class, interaction and package diagrams together

Design model is a term from the Rational unified process (cf. section 4.3). A design model encompasses the static structure and the dynamic behavior of the future system. The structure is specified through design classes assigned to packages (subsystems), with well-defined interfaces and relationships. The behavior of the system is described with the help of interaction diagrams. In UML, class, interaction and package diagrams together form a design model.

Package diagrams

Interaction diagrams and design class diagrams are explained in more detail below. A *package diagram* groups UML elements that belong together, in particular classes, interfaces and/or other packages (i.e. nested packages). In this way, large systems can be organized into subsystems. Packages are also used in architectural design to organize layers of a software system, assembling all classes, interfaces and packages that make up a layer into one package. Figure 5-26 (cf. section 5.1.4 above) contained an example of a package diagram showing a logical architecture with three layers.

Interaction diagrams

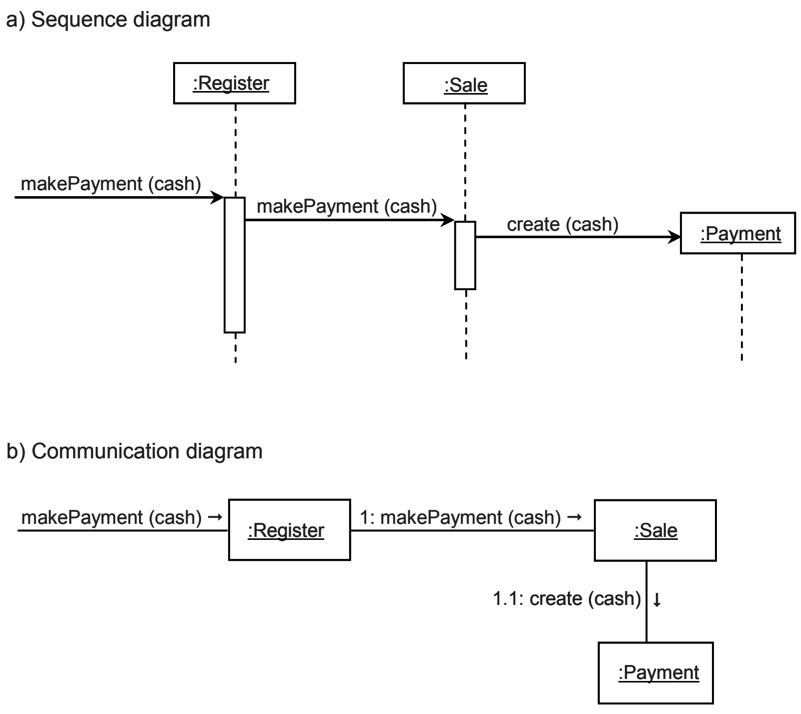
Sequence and communication diagrams

UML provides two types of interaction diagrams: sequence diagrams and communication diagrams. Both serve the purpose of illustrating how objects are connected, i.e. how they collaborate by exchanging messages with each other.

Messages

Messages are written on the lines connecting the objects. UML message syntax specifies that messages must have a *name* and that they may have *parameters* (possibly with parameter types) and a *return type* (type of the result returned). In the example of figure 5-30, "makePayment" is a message name and "cash" a parameter (whose type is not explicitly noted because it is obviously some currency type).

Figure 5-30 **Excerpts from sequence and communication diagrams**§



§ Larman 2005, pp. 224-225.

Classes and instances

Classes and objects are drawn as rectangles with the name written inside. Interaction diagrams usually show instances (objects) and not types (classes). To notate an *instance*, the name is preceded by a colon (which may also be preceded by a unique identifier). For example:

`Sale`	is the "Sale" class,
`:Sale`	is an object of the "Sale" class (an instance),
`dom:Sale`	is an object of a particular "Sale" class, uniquely identified through the name "dom".

Sequence diagrams

Sequence diagrams may consume much horizontal space, because they extend towards the right every time an object is added. The horizontal axis represents the flow of time, meaning a sequence diagram is basically read from left to right and top-down.

The upper section of figure 5-30 shows part of a sequence diagram for a scenario of the "Process sale" use case of figure 5-6. The message "makePayment(cash)" is received by a "Register" object which now sends a "makePayment(cash)" message to a "Sale" object. This object sends a "create(cash)" message to a "Payment" instance etc.

The vertical bars interrupting dashed lines are called *execution specification* bars (previously called activation boxes). They show where the focus of control is.

Communication diagrams

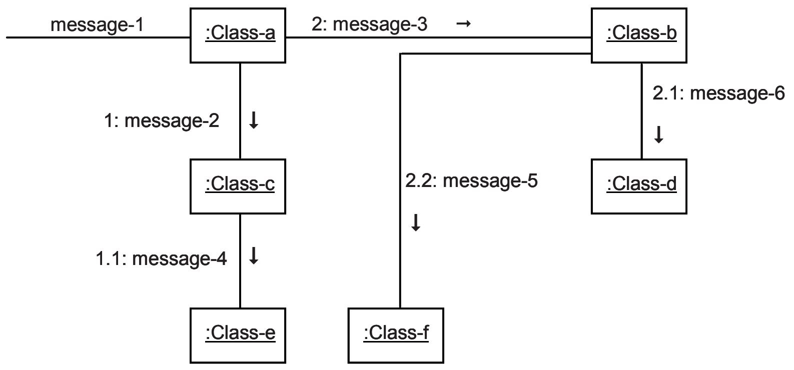
The same problem is described with the help of a *communication diagram* (previously called collaboration diagram) in the lower part of the figure. The sequence of messages is now not simply from left to right and from top to down as in a sequence diagram. Instead, messages have labels indicating the sequence and the nesting of messages. In figure 5-31, the nested labels define the sequence of messages as:

1. message-1, invoking <u>:Class-a</u>
2. message-2, invoking <u>:Class-c</u>
3. message-4, invoking <u>:Class-e</u>
4. message-3, invoking <u>:Class-b</u>
5. message-6, invoking <u>:Class-d</u>
6. message-5, invoking <u>:Class-f</u>

Communication diagrams are more compact than sequence diagrams, requiring less space since objects/classes can be drawn anywhere when the diagram is being developed. In a sequence diagram, new objects/ classes always extend the diagram to the right.

Interaction diagrams can be extended and refined with many details, especially with control structures such as loops and conditional messages and advanced object-oriented concepts such as polymorphism and asynchronous messages. Reader interested in more information are advised to consult the UML specification [Booch 2005] or dedicated UML books [e.g. Larman 2005].

Figure 5-31 **Examples of nested labels in a communication diagram**



Design patterns

GoF patterns ("Gang-of-Four patterns")

For many recurring design problems, solutions have been described in the form of design patterns. As many authors have developed such patterns, hundreds of them are available today. The seminal work of Gamma, Helm, Johnson and Vlissides [Gamma 1995] defined 23 design patterns which are nowadays known as the *GoF patterns* ("Gang-of-Four patterns"). Some of the problems addressed in the GoF patterns occur quite often, others are less common.

The GoF patterns are subdivided into three categories:

– Creational patterns – handling the instantiation process (how, when, and which objects are created)
– Structural patterns – handling the composition of classes and objects, how classes and objects are used in larger structures, and how interfaces are separated from implementation
– Behavioral patterns – handling the communication between objects and the responsibilities of objects

Widely used GoF patterns include the following [Gamma 1995, chapters 3-5]:

– *Factory method* (creational): Sometimes it is useful to have a mechanism for instantiating objects although the class of the object is not known yet. The factory method pattern helps the designer to model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate. This means that the creation of objects is deferred to the subclasses.
– *Abstract factory* (creational): An abstract factory is an interface or an abstract class for creating families of related or dependent objects (e.g. buttons, listboxes etc. on a graphical user interface) without requiring the developer to specify the concrete classes (e.g. Windows button, Java Swing button, MacOS button) of these objects in the definition of the abstract factory. An abstract factory has a method that returns one of several concrete factories. Each concrete factory can instantiate on request several different objects as declared in the definition of the concrete factory.
– *Singleton* (creational): A pattern to make sure that only one object of a class can be instantiated and that this object can be accessed from anywhere (global visibility).
– *Adapter* (structural; also known as *wrapper*): Converts the interface of a class into another interface that is expected by the client class. Adapters let classes work together that would otherwise not be able to because of incompatible interfaces. The client invokes the adapter's methods instead of calling directly the methods of the incompatible class.
– *Composite* (structural): The composite pattern lets other classes treat individual objects and compositions of objects in the same way (e.g. polymorphically). Both the individual objects and the composite implement the same interface. Thus a client call to an interface method does not need to differentiate whether the object invoked is an atomic object or a composite object.
– *Façade* (structural): Provides a unified interface to existing implementations or interfaces of a subsystem; in other words, this pattern puts up a new interface (a façade) in front of the original subsystem, hiding internal details. The façade pattern makes it possible to use

the façade like a black box that provides services. It defines a higher-level interface, making the subsystem easier to use§.

– *Proxy* (structural): This pattern helps to control access to an object with a proxy (also known as a surrogate or placeholder), e.g. when direct access to the object is not possible or not desired. A variant is the *remote proxy* pattern in which a local proxy (called a "stub") provides services that are actually performed by a remote object.
– *Mediator* (behavioral): Defines an object that is responsible for controlling and coordinating the interactions of a group of other objects. In this way, collective behavior is encapsulated in a separate mediator object. The mediator pattern promotes loose coupling by keeping objects from referring to each other explicitly. Objects communicate with the mediator and not directly with each other.
– *Strategy* (behavioral): This pattern is useful for situations where it is necessary to dynamically select or swap the algorithms used in an application. It defines a family of algorithms, encapsulates each one in a separate class, and makes them interchangeable. The strategy pattern makes it possible to change an algorithm independently from other objects that use it.

GRASP patterns

Another approach to defining patterns, based on fundamental object-oriented design principles, was proposed by Larman under the name *GRASP*, initially an acronym for "General responsibility assignment software patterns". GRASP patterns are more fundamental than the GoF patterns, describing fundamental principles of software design and not focusing too much on solutions to particular problems.

Key notion is responsibility

The key notion in GRASP is *responsibility*, an important concept used in a design approach called responsibility-driven design (RDD). Essentially, responsibility means that an object is responsible for doing or for knowing something [Larman 2005, p. 276]:

– Being responsible for *doing* something includes doing it alone (such as creating an object or doing a calculation), initiating an action in other objects, and controlling and coordinating activities in other objects.
– Being responsible for *knowing* something includes knowing about private internal data, knowing about related objects, and knowing what the object can derive or calculate.

§ Façades are used, for example, to encapsulate legacy system functionality to make it accessible through web services in modern service-oriented architectures [Jankowska 2005, Kurbel 2006]; cf. section 7.4.3.

Based on these types of responsibilities, Larman describes nine design patterns called information expert (an object that knows something), creator (an object that does something), controller, low coupling, high cohesion, polymorphism, pure fabrication, indirection and protected variations. Interested readers can find details of these patterns in Larman's book [Larman 2005, pp. 281-318, 413-434]. One thing about the GRAPS patterns is worth noting: Understanding responsibilities helps very much in creating good object-oriented designs!

Documenting patterns

Patterns can be documented in plain text, in a semi-formal notation such as UML diagrams, or in both. In the GoF book, the pattern structure is described with a class diagram, sometimes supplemented by an object diagram (showing instances) and a sequence diagram. Most aspects of a pattern such as intent, motivation, participants (classes, objects), applicability (to which situations), implementation aspects and sample code are explained in several pages of text. The GRASP patterns are documented in a similar way.

Class diagram for proxy pattern

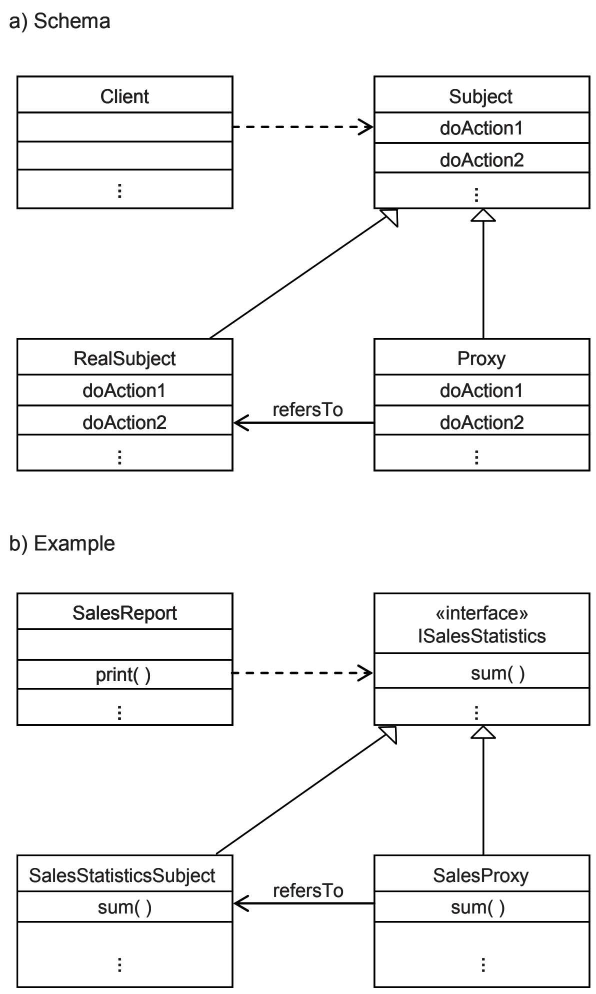
Although many types of UML diagrams can be used to describe a pattern, class diagrams and interaction diagrams are the most common ones. Figure 5-32 a) shows, as an example, a class diagram for the *proxy pattern* [Gamma 1995, p. 209, Eriksson 2004, p. 261]. Apart from the client class invoking interface operations, another three classes are involved in this pattern: a "RealSubject" class, a "Subject" class, and a "Proxy" class.

The "Subject" class provides an interface (operations) that clients can use. This interface is implemented in the "RealSubject" class (possibly far away on a remote computer) and also in the "Proxy" class. However, the "Proxy" class only delegates any calls it receives to the "RealSubject" class. A "Client" class object always works, through the "Subject" interface, with the "Proxy" object, so the "Proxy" object controls access to the "RealSubject" object[§].

An example illustrating the schema of figure 5-32 a) is given in the lower half of the figure. Suppose a "SalesReport" object (client) needs a sales total from a particular time period, which can be computed by a "sum()" operation of the "SalesStatisticsSubject" class.

§ The diagram uses a notation similar to UML design class diagrams which are explained later in this section. An arrow with a large hollow arrowhead indicates implementation of an interface here, an arrow with a regular arrowhead is an association. A dashed arrow line is used by Gamma et al. to indicate that an external client initiates the creation of the objects in order to realize the pattern [Gamma 1995, p. 364].

Figure 5-32 **Proxy pattern described in a class diagram**[§]

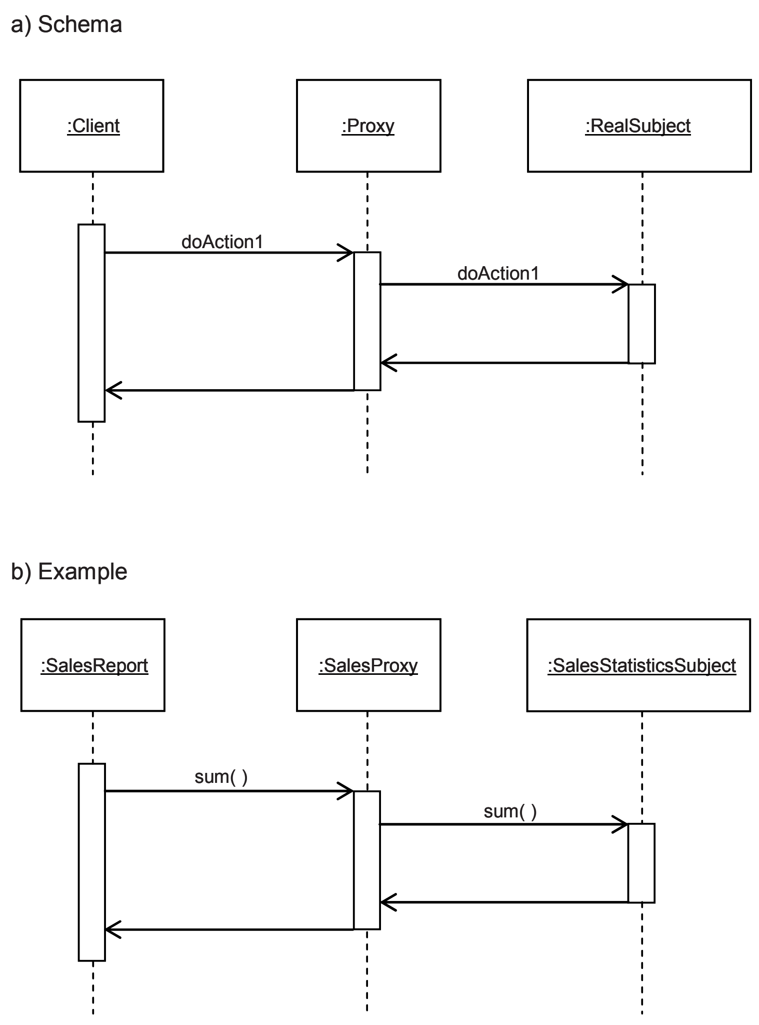


However, the client object actually invokes the "sum()" operation which is provided by the interface "ISalesStatistics" and implemented in the

§ Based on: Eriksson 2004, pp. 261-266.

"SalesProxy" class. This implementation is basically a call to the "sum()" operation of the "SalesStatisticsSubject" instance (possibly a remote method invocation over a network).

Figure 5-33 **Proxy pattern described in a sequence diagram**



Proxies often do more tasks than just forwarding requests, e.g. performing local tasks which do not require the complete functionality of the "RealSubject". In this way the performance of the system can be enhanced.

Sequence diagrams for proxy pattern

Sequence diagrams for the proxy patterns of figure 5-32 are drawn in figure 5-33. In the application example in part b) of the figure, a client instance (a "SalesReport" object) sends a "sum()" message to the "SalesProxy" object implementing the "ISalesStatistics" interface. The "Proxy" object sends itself a "sum()" message to the "SalesStatistics-Subject". This object will respond to the request of the "Proxy", and the "Proxy" object can now return the result to the "SalesReport" object that started the inquiry.

Design class diagrams

The second major outcome of design activities, next to interaction diagrams, are class diagrams. Patterns can help find the appropriate classes for many stereotypical problems. Class diagrams were introduced in section 5.1.4 where they served to represent domain concepts in requirements engineering. The same diagrams, with more detail, are used in design to represent design elements such as classes which are meant to be implemented as program code later.

Class diagrams: domain concepts in RE, software entities in design

In order to distinguish a class diagram for design from a class diagram representing a domain model, the former type is often called a *design class diagram (DCD)*. The visual symbol for classes contains three compartments: classifier name, attributes and operations.

Classifier name compartment

The *classifier name* in a DCD is usually either the name of a class or the name of an interface. (An interface specifies behavior, usually by defining abstract operations that classes supporting the interface have to implement.) In a class diagram, interfaces are marked with the stereotype «interface». The classifier name can be qualified if the class or interface belongs to a package, e.g. "sales.marketing.Invoice". By convention, class names and interface names are capitalized.

Attributes compartment

Attributes describe characteristics of the objects. In design classes, attributes are defined with data types. Common types (e.g. primitive types such as number) are written directly with the attribute name. For abstract types (reference types), UML allows the developer to write the type (class name) following the attribute name and a colon, or to draw an association line pointing to the respective class, or both. In figure 5-34, for example, the data type of the "customerToBill" attribute is "Customer", and an association line is used to reference the "Customer" class. (Note that "Employee" and "IDType" are also reference types whose classes are not shown in the figure.)

Operations compartment

Operations are written in the third compartment. An operation has a name and it can have parameters, a visibility (e.g. private, public) and properties (additional information, e.g. exceptions that may be raised).

An operation creating an instance of the class is prefixed with the stereotype «constructor». For the "invoiceItem" class, two constructors were explicitly modeled. The first one, "InvoiceItem (advertisement, edition, price)", creates an invoice-item line. For this purpose, it has to get the relevant advertisement, the edition and the price of the publication. The second one, "InvoiceItem (publ)", also creates an invoice-item line, but from information provided by "publ" (a "Publication" object).

The example used in figure 5-34 to illustrate design class diagrams is similar to the example modeled in a previous entity-relationship diagram (cf. figure 5-25). It is about a newspaper company selling advertisement space in their newspapers. The same advertisement can be published in different issues of the newspaper. Customers are invoiced for individual publications, depending on their status as agencies or regular customers.

Associations in design class diagrams connect design classes. Associations have names, multiplicities and sometimes small triangular arrows indicating the reading direction (e.g. "Customer Orders Publication" in figure 5-34).

Attribute declaration referencing another class

When an *attribute* declaration references another class (using the class name as its data type), an arrow (called navigability arrow) shows the direction from the source to the target. A role name is used to express the role played by the referenced class in the specific context of the association (it is common to use the name of the attribute as a role name). The role name and the multiplicity are placed only at the target end of a navigability arrow. In the figure, the "Invoice" and "Customer" classes are associated via the "customerToBill" attribute referencing the "Customer" class. The association line shows the role "customerToBill" and the multiplicity "1" (i.e. the invoice goes to exactly one customer).

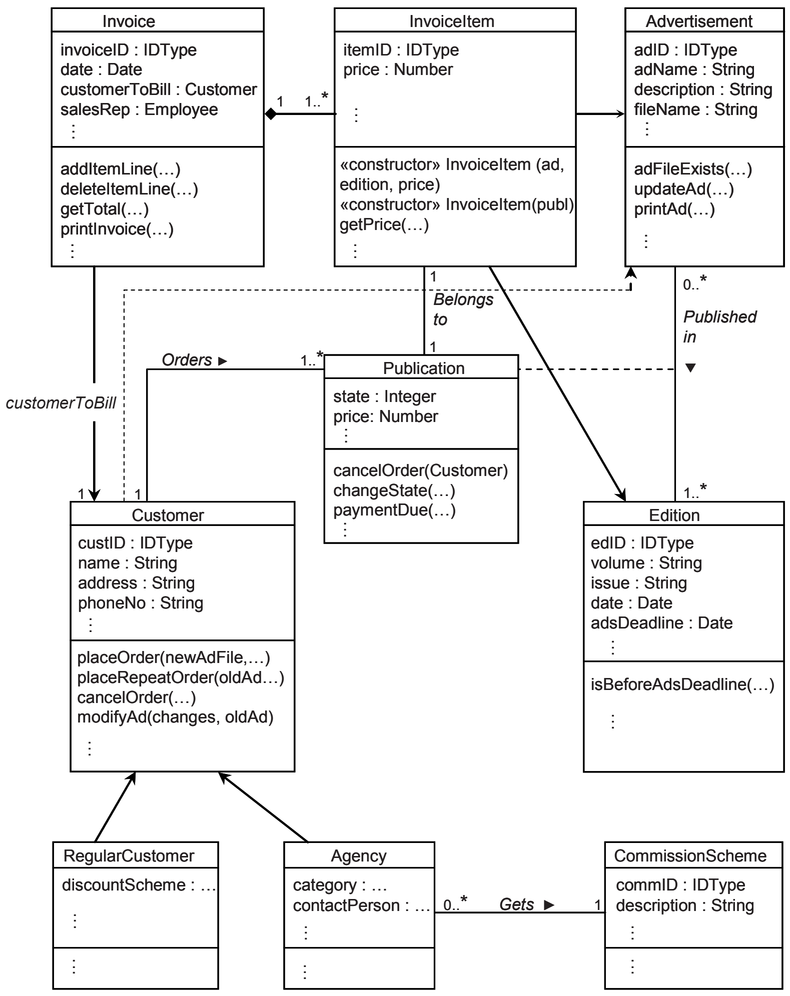
Dependency relationship

A *dependency relationship* in UML is a relationship between two classes which is characterized by the fact that one class (the consumer or client) has knowledge of some matters of the other class (the supplier). This means that a change to the supplier class may require a change in the client class.

Indirect dependency via parameter types

Dependency is a very broad concept, including relationships that have their own representations in UML. For example, any association is a dependency, expressed by an a association line. Therefore dependency lines are only needed for those types of dependencies that do not have their own representations. An example is an indirect dependency via parameter types.

Figure 5-34 **Design class diagram (example)**



Suppose a method of class A receives a parameter of type B, and B is a reference type defined by class B. Then A depends on B because if B's interface is changed, A's implementation may also need to be changed. Dependency relationships are shown as dashed arrow lines.

In figure 5-34, a dependency relationship exists between the "Customer" and "Advertisement" classes, because the "placeRepeatOrder" method of "Customer" uses a parameter "oldAd" whose type is "Advertisement".

Dependency relationships include generalization as introduced in section 5.1.4 and using (i.e. implementing) an interface. UML provides several notations to describe an interface implementation. As a dependency relationship, it is drawn as a dashed arrow with a large hollow arrowhead. Interfaces were shown in the proxy pattern description above, however, they followed the style of the original authors (i.e. solid instead of dashed lines).

Composition (composite aggregation)

Composition is an association type in which a whole is composed of parts. Since the whole is aggregated from the parts, this association is also called *composite aggregation*. The parts exist only within the whole, and the whole without the parts does not exist either. For example, an "InvoiceItem" object as in figure 5-34 can only exist if an "Invoice" object exists and vice versa (an invoice without items makes no sense). A composition is marked with a filled diamond on the association line.

Association classes

An *association class* serves the same purpose as a re-interpreted relationship type in the entity-relationship model explained in section 5.1.5: Sometimes it is necessary to treat an association as a class, for example if it needs to have its own attributes, operations and associations with other classes. In the figure, "Publication" is an association class, connected with the classes "InvoiceItem" and "Customer".

Singleton class

A *singleton class* – a design pattern mentioned above – has at most one instance. This fact can be indicated by writing a "1" in the top right corner of the classifier compartment.

User-defined compartments

User-defined compartments may be added to the default compartments classifier name, attributes and operations. A user-defined compartment is sometimes introduced to describe exceptions that can be raised by objects of the class.

Object diagrams

The same notation as for class diagrams is used for *object diagrams*. An object diagram shows specific instances of classes and specific links between those instances at some point in time ("a possible snapshot of the system's execution" [Eriksson 2004, p. 25]). To distinguish object names from class names, object names are written the same way as described above in the part on interaction diagrams, i.e. preceded by colon and underlined.

More DCD elements

UML design class diagrams can have a number of additional elements such as refinements (introduced in section 5.1.4), or-associations (mutually exclusive associations), constraint associations (one association is a subset of another association) and derivations (rules how to derive something). Discussing all facets of class-diagram notation is beyond the scope of this book. Interested readers can find more details in UML 2 references, guides and dedicated books [e.g. Booch 2005, Larman 2005, Eriksson 2004].

Activity diagrams

Activity diagrams are a popular means to model processes. An activity diagram shows the steps of the solution and the flow of control between the steps. Not only people used to procedural programming and hence to procedural design like activity diagrams. Object-oriented developers also appreciate them because they provide a richer notation of activity sequences than communication and sequence diagrams. For example, they let the designer describe conditional execution and refinement of an activity, events triggering an activity and organizational aspects (e.g. who performs the activity, where is it performed).

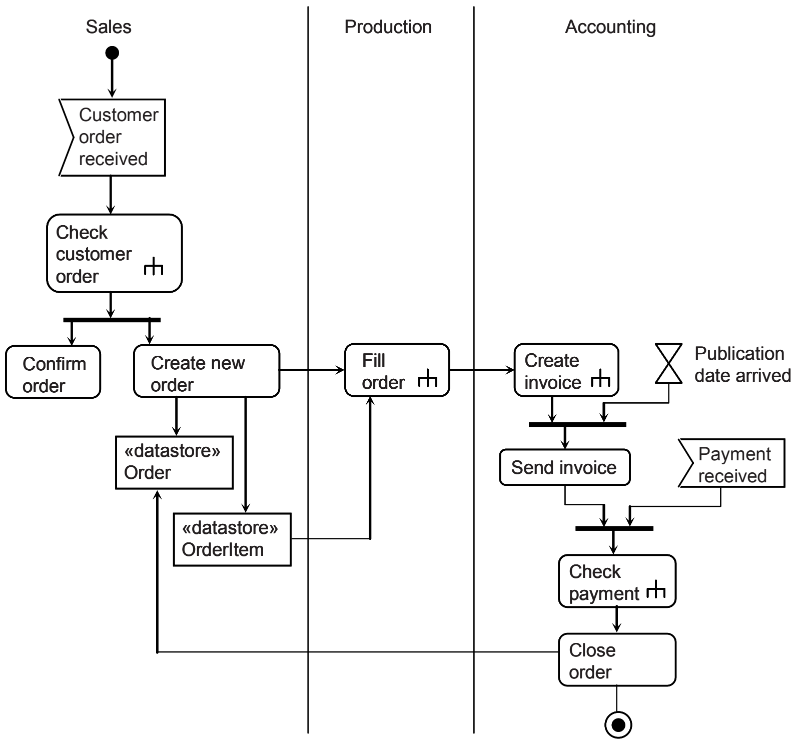
Activity diagrams are similar to data flow diagrams (DFDs) in structured analysis (SA), which were discussed in section 5.1.5. However, DFDs in SA are primarily used for capturing, analyzing and documenting requirements. UML activity diagrams, on the other hand, are mostly used in design, but they can also help to model complex requirements (in addition to use cases) and implementation details.

Main elements of activity diagrams

The main elements of an activity diagram are actions, transitions and object nodes. The flow of control is described with the help of arrows, forks, joins, rakes and signals. Organizational aspects are expressed by swimlanes. A solid circle and a bull's eye circle indicate the start and the end of an activity diagram, respectively. Among the most common elements and symbols in activity diagrams are the following ones (some of them are also shown in the figures 5-35 to 5-37):

- *Action* – does something in order to produce a result. The action symbol is a rectangle with rounded corners. In figure 5-35, "confirm order", "fill order" etc. are actions.
- *Transition* – connects actions, indicated by an arrow. Upon completion of an action, transition to the next action(s) happens automatically.

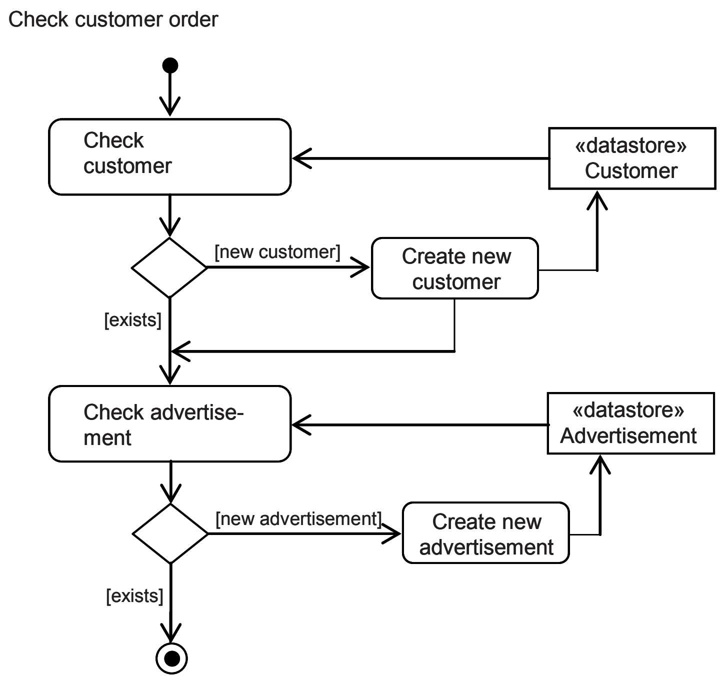
Figure 5-35 **Activity diagram for customer order process (example)**



- *Object nodes* – represent objects either produced by the actions or needed to perform an action. An object node is represented by a rectangle. Object nodes serving as data stores as in SA (cf. section 5.1.5) are marked with the stereotype «datastore». For example, "Order" is an object node (a data store).
- *Fork* – splits one transition into two or more (i.e. several parallel actions follow). The symbol for a fork is a bold line. In figure 5-35, a fork splits the transition going out of "Check customer order" into two transitions, one towards "Confirm order" and another one towards "Create new order".

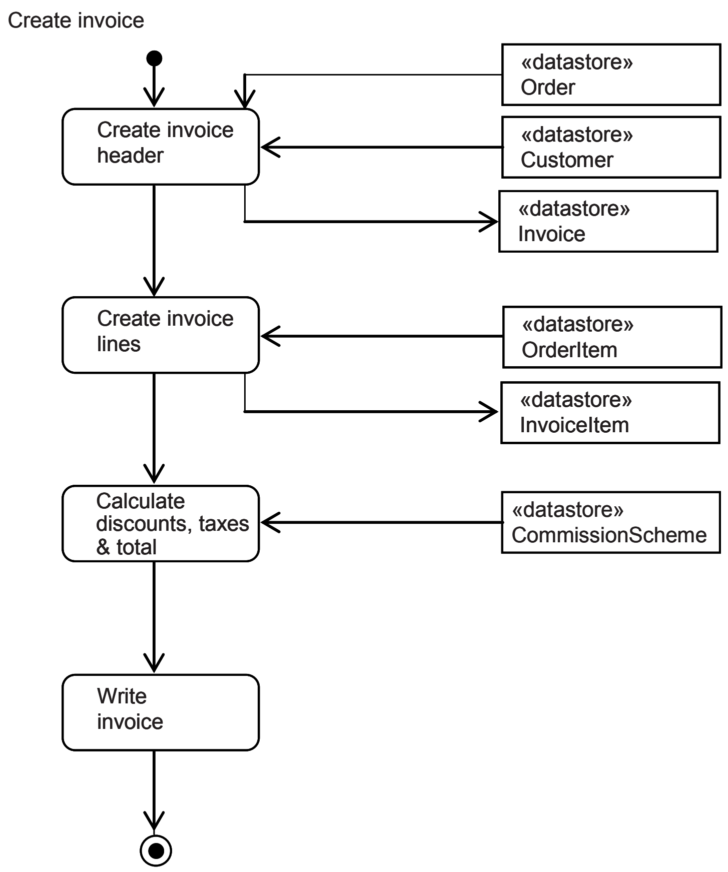
- *Join* – unifies several branches, making one transition out of multiple transitions. The symbol is the same as for a fork, but it has multiple ingoing lines and only one outgoing line. A join is used, for example, on the right-hand side of figure 5-35 to make sure that both an invoice has been created and that the publication date has arrived before the "Send invoice" action is performed.
- *Decision* – specifies mutually exclusive branches. Depending on a condition, one of the branches is selected. A diamond is used to indicate a decision. Branches can be labeled as in figure 5-36 where "[exists]" and "[new customer]" are labels.
- *Merge* – brings branches back together. It has the same symbol as a fork, yet with two or three ingoing transitions and only one outgoing transition.

Figure 5-36 **Expanded activity diagram for "Check customer order"**



- *Rake* – indicates expansion of an action into a more detailed diagram, i.e. the action is refined in the second diagram. Figure 5-35 contains four actions that will be expanded in separate diagrams. Two of these are shown in figures 5-36 ("Check customer order") and 5-37 ("Create invoice").

Figure 5-37 **Expanded activity diagram for "Create invoice" action**



- *Signals* – can be received and sent from inside or outside the diagram. They can be used to model events that trigger an action. Signals are pentagon-shaped and come in three variants; one for sending a signal, one for receiving a signal, and one for a timer signal (received when a set time interval is over or a set time arrives). Examples shown in figure 5-35 are "Payment received" (a signal received) and "Publication date arrived" (a timer signal).
- *Swimlanes* – partition an activity diagram into multiple activity sequences. Any suitable criterion may be chosen to divide an activity diagram into swimlanes; for example, responsibility for an action. Swimlanes can thus be used to show different actors or organizational units in a business context. In figure 5-35, there are three swimlanes representing the "Sales", "Production" and "Accounting" departments.

A simplified order process

The sample diagrams in figures 5-35 to 5-37 show a simplified order process related to the class diagram of figure 5-34. When the newspaper publishing company receives a customer order stating that the customer wishes to have one or more advertisements published, the order must be verified first: Does the customer already exist in the company's database? Are the advertisements already available in the advertisements database? If not, then a new customer record and/or one or more new advertisement records have to be created. When everything is OK, the order is confirmed to the customer and a new customer order is created, leading to an additional order record and one or more new order item records in the database.

While the initial actions are performed by the "Sales" department (cf. figure 5-35), completing the order (i.e. placing the advertisements in the correct newspaper issues, pages and locations) is under the responsibility of the "Production" department. The "Accounting" department handles invoice creation. The invoice contains the order and customer data. The total is computed taking customer discounts and taxes into account; afterwards the invoice is stored in the database and formatted for printing. When the payment arrives, the amount received needs to be checked against the invoice details. At the end, the order is closed.

Frameworks

A framework is a reusable design

Frameworks can help significantly to reduce the design and implementation effort, provided that a framework for the problem on hand is available. Generally speaking, a framework is a reusable design for a software system or a subsystem, made up of a number of modules that can be extended by other, application-dependent modules.

Frameworks are available for various problem categories such as graphical user interfaces, middleware and enterprise applications. Examples include the following:

– GUI frameworks such as Java Swing, a framework for creating interactive Java systems [Zakhour, ch. 15], and Apache Struts, a free open-source framework for creating Java web applications [Apache 2007]. Both are, by the way, based on the model-view-controller architectural pattern.
– Middleware frameworks such as JNI (Java native interface), Java RMI (remote method invocation) and the Microsoft .NET framework.
– Enterprise application frameworks such as JBoss Seam and SAP Composite Application Framework.

Enterprise application frameworks

Enterprise application frameworks support the development of information systems in a business domain. Such frameworks can be specialized in certain application areas, e.g. financial engineering, or in the application of certain technologies. JBoss Seam, a framework that supports Java EE web development [Yuan 2007], and the "SAP Composite Application Framework" [SAP 2007b] are examples of the second category. The SAP framework provides an environment for the design and use of composite applications within an enterprise service-oriented architecture (ESOA) as discussed in section 3.4.

GUI frameworks

The most common framework type defines frameworks for *graphical user interfaces*. In all windowing systems, the same "infrastructural" problems occur, such as creating, resizing, moving, and closing a window; handling user actions (button click, listbox selection, etc.); or navigating between web pages. Suppose the system under design is an electronic web shop. If a GUI framework is available, the development team can concentrate on the application problem (e.g. registering customers, displaying the product catalog, managing the shopping cart, creating and processing customer orders etc.) instead of worrying about the "plumbing" behind it (e.g. basic form handling, error-free navigation from one page to another, mechanisms for database access).

Frameworks are in principle not tied to a particular paradigm (such as object-oriented analysis and design), but nowadays they are usually realized with the help of object-oriented concepts (such as abstract and concrete classes, interfaces and objects). In object-oriented terms, a framework can be defined as follows:

Definition: framework

> A *framework* consists of a cohesive set of abstract and concrete classes and interfaces for a particular problem type, including a mechanism to plug in additional classes and to customize and extend the provided classes. The framework also specifies what the framework user has to provide through abstract classes and interfaces.

Frameworks provide core functionality

In other words, a framework provides core functionality for a certain problem category and a way to adapt and extend the core functionality according to the needs of the particular application problem. This means that a framework is usually not the same as the final software system solving the application problem but a powerful means to create such a system. Users of the framework (designers, developers) create the application-dependent components and satisfy requirements expected by the framework components; for example, designing and coding subclasses of abstract classes, implementing interfaces and handling exceptions.

System structure in-the-large

A framework determines the system structure in-the-large, i.e. how the system is subdivided into classes and objects, how the classes and objects collaborate, and the flow of control between them. The major design decisions valid for all application systems in the respective domain were already made by the framework designers. The project team can concentrate on the application-specific requirements of the information system under development. The major part of the final solution consists of the framework components, and only the application-dependent components have to be written by the developers.

"Don't call us, we'll call you"

In contrast to class libraries and packages that also provide predefined components, frameworks play an *active* part. Whereas the invocation of library and package classes is under the control of user-written modules, frameworks rely on the *Hollywood principle*: "Don't call us, we'll call you." [Larman 2005, p. 627] It is the framework objects that control the flow of actions, not the user-written objects. The latter ones have to react and act according to the framework's calls. For example, framework classes send messages to user-written classes, expecting in the first place that the class has been implemented and secondly to receive an appropriate response.

Design patterns and frameworks

Frameworks and design patterns are similar concepts which are often used together. Embedding design patterns in a framework can significantly increase the degree of reusability of the framework. The more design patterns included in a framework, the more general the framework becomes.

Since frameworks and design patterns resemble each other, we will point out the differences between the two concepts. According to

Gamma et al., there are three major differences distinguishing frameworks from design patterns [Gamma 1995, p. 28]:

Frameworks are designs plus code

1. Design patterns are more abstract than frameworks. Design patterns are basically textual descriptions of problem-and-solution categories, possibly explained with the help of some sample code. Nevertheless a pattern has to be implemented anew each time it is used in a concrete application context. Frameworks, on the other hand, are designs plus code. While the framework design is documented in text or another appropriate notation, many classes, interfaces and objects are available as code written in a programming language such as Java or C++.
2. Design patterns are smaller than frameworks. A framework can contain a number of design patterns but a design pattern does not contain frameworks.
3. Design patterns are more general than frameworks. A framework always has a specific application domain, while most design patterns are applicable in any domain as long as the general problem type underlying the pattern occurs in that domain. For example, an abstract factory may be useful in a GUI framework as well as in a financial-accounting framework.

Advantages: reusability and less development effort

The main advantages of frameworks are reusability and lower development effort, since essential design decisions have already been made and part of the implementation is already available. On the other hand, the creativity of the designers and the degree of freedom to make choices are restricted by the framework.

Since the framework does most of the work and additional components must comply with the framework, a framework upgrade usually has an impact on the application modules, meaning that the user-written components may need to be modified as well. Loose coupling with the framework components is thus an important aspect in designing user-written components.

Business frameworks are heavyweight

Using a business framework can significantly improve development productivity. On the other hand, business frameworks are heavyweight products, requiring a lot of time to get acquainted with. In a large organization, the benefits will usually outweigh the cost. However, the cost and learning efforts can be prohibitive for a small company.

5.2.3 Designing the User Interface

Software systems are *used* to solve problems. Many software systems, such as most business information systems, are used by humans. However, software systems can also be used by machines or other technical devices, e.g. embedded systems software.

Interfaces for human users

User-interface (UI) design is about interfaces for software systems that have *human users*. The interface between the human user and the software is of considerable importance in the development of an information system. There are two main reasons for this: 1) For the user, it is the interface that exposes the system's functionality, and 2) the user *interacts* with the system through the interface.

General principles of UI design

Human computer interaction (HCI), software ergonomics

A general requirement for UI design is that the interface should support users in their work as effectively and efficiently as possible. From the 1980s on, researchers have investigated how to do this. Even new academic disciplines such as *human computer interaction (HCI)* and *software ergonomics* came into existence. Annual conferences dealing with these topics show that there is ongoing interest in both the academia and in practice (for example, the annual HCI International Conferences; http://www.hci-international.org/).

General principles for user-interface design have been developed by many researchers and practitioners over the years. We will not discuss in detail this research and instead refer the reader to the relevant literature [e.g. Shneiderman 2005]. An aggregated list of UI design principles compiled by Sommerville is shown in figure 5-38.

Usability

The goal of user-interface design is to enhance the *usability* of the underlying software system. Usability, as a non-functional requirement for software systems, was mentioned above in section 5.1.1. It expresses how easy or difficult it is for the user to work with the system and to achieve whatever the intent of using the system is.

Utility

Usability depends primarily on the quality of the user interface. Usability should be distinguished from *utility*, which refers to the expected functionality, i.e. does the system do what the user needs? Jakob Nielsen, who has been called the "usability pope", characterizes *usability* by five quality components as shown in figure 5-39: learnability, efficiency, memorability, errors and satisfaction [Nielson 2003].

Figure 5-38 General principles for user-interface design§

Principle	Description
User familiarity	The interface should use terms and concepts drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of the system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system users.

User interfaces depend on the device

By their nature, user interfaces can be quite different, depending on the type of device they are displayed on. A typical business information system interfaces with the user via a computer monitor, plus a keyboard and a mouse as input devices. However, some types of systems use different devices to display information – for example small LCD screens on mobile phones and voice generators in navigation systems. All these types of systems require an interface design. Although in this section we are not explicitly focusing on a particular type of system, an implicit assumption is that the user has a regular computer monitor, a mouse and a keyboard. Nevertheless, the basic principles apply to other types of user interfaces as well.

§ Sommerville 2007, p. 364.

Figure 5-39 **Characteristics of usability [Nielsen 2003]§**

Learnability	How easy is it for users to accomplish basic tasks the first time they encounter the user interface?
Efficiency	Once users have learned the user interface, how quickly can they perform tasks?
Memorability	When users return to the interface after a period of not using it, how easily can they reestablish proficiency?
Errors	How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
Satisfaction	How pleasant is it to use the interface?

Graphical user interface (GUI)

Most information systems today have a *graphical user interface (GUI).* The term GUI design has more or less become a synonym for user-interface design. Graphics have become increasingly popular, not only as a general basis for interaction with the user but also as means to present compiled information. For example, managers appreciate graphical systems such as dashboards and cockpits because they are very effective means to present and manipulate management-relevant information.

Visualizing data

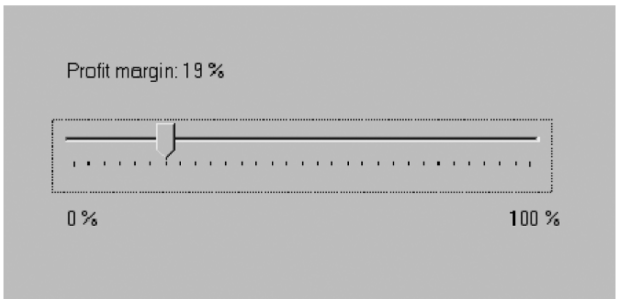
Graphics have many *advantages* over text. Visualizing data can help the viewer to get relevant information out of the data faster and more efficiently. In many situations, not the detailed data items or the precise numeric values are important but the general picture behind them. For example, a sales manager interested in the development of the main competitors' market shares will be served better with pie charts or with a stacked-bars chart than with numbers like 13.86 %, 22.51 % etc.

Graphical information displays can also be used to change or enter numerical values. In the example of figure 5-40, a horizontal trackbar is

§ In Nielsen's document, the term "design" is used instead of "user interface" or "interface" as in the table. We made this change because the meaning of the word "design" is not the same as the meaning we are using in this chapter.

used to display the extent to which a numerical goal (profit margin) has been achieved. In addition, the user can move the slider in order to set a new value for the profit margin from which other parameters are recomputed.

Figure 5-40 **Graphical display and manipulation of a numerical value**



Disadvantages

Graphics are usually more pleasant to look at than text and numbers, and sometimes they are more efficient to show and manipulate information. However, there are also *drawbacks* and situations where textual representations have advantages over graphical ones. Graphics can take up a lot more screen space and require more time to download than text. For example, a list of quarterly sales figures can be represented in one or two short lines of text (e.g. name of the month in one line, sales figure underneath), whereas the same information shown as a bar chart would occupy a significantly larger portion of the screen. Graphical representations provide a quick visual impression of matters while text is better to present details and exact numerical values.

Colors

Colors are popular means to associate a meaning or importance with information. Colors are often used to express the significance or the status of things. In a dashboard, for example, red color could stand for "high risk", yellow for "observe this" and green for "OK". However, colors should better be used carefully and conservatively. The reasons for this are: 1) The same color may mean different things to different persons; 2) some people have problems to distinguish between certain colors or are completely color-blind, and 3) color mixes created by amateurs (i.e. not by professional graphics designers) often look unprofessional.

Types of users

The user interface should meet the user's needs. But who is "the user"? The same information system can have very different types of users. Common categories – somewhat overlapping – are the following:

User categories

- » Expert users – persons who are very experienced in what they are doing with the system
- » Power users – persons who exploit significant portions of the system's functionality in their work
- » Novices – people who are new to the system, learning how to use it effectively
- » Permanent users – people who use the system every day, for example to do their daily work
- » Casual users – people who use the system every now and then

Different user types have different needs

Obviously, the needs of these types of users are quite different. Novices, for example, need easy, user-friendly access, navigation, support and help features. Power users, on the other hand, want to get their work done as efficiently as possible. Where a novice user might prefer step-by-step navigation through self-explanatory menus and listboxes, power users would consider this type of support rather in their way. They might be happy with shortcuts or with typing abbreviations because they know exactly where to go and how to get their work done. Casual users are again different. Someone who uses the system only once in a month has to receive more support and guidance than someone who works with the system every day.

Categories of users according to their roles

Another differentiation of users is according to their *roles* as users of the information system. Typical business information systems have categories of users such as the following:

- Customers
- Suppliers
- Employees: clerical workers, knowledge workers, managers
- System administrators

All users work with the same information system, yet with different, possibly overlapping parts of the system and on different levels of proficiency. *System administrators* are usually power users. A *customer* can be a casual user, placing an order with the help of the system's web front-end every now and then. However, the customer can also be a permanent user, placing several orders every day (because he or she is an industrial buyer working in a procurement department). *Employees* processing the daily orders are permanent users requiring efficient access to customer, order and production data. This can best be provided through a forms-based interface including data-handling tools. *Knowledge workers* analyzing customer behavior with the help of business-intelligence tools need an effective interface to handle large amounts of data, in particular numbers, and graphical tools for visualization.

Non-trivial IS have multiple types of users

These sample descriptions show that user categories can be quite heterogeneous. Most non-trivial information systems have multiple types of users. System designers need to take this variety into account when designing the user interface. Consequently, a system can have more than one user interface, or at least several variants of the interface.

In addition to multiple user types, different technical devices used as front-ends can also call for several interfaces. Suppose the requirements specification states that access to the information system must be provided not only from computer monitors but also from PDAs and UMTS phones. This means that the same information, produced by the system, must be made available in different forms.

Model-view-controller pattern

An approach to separate the presentation of information to the user from the information itself is provided by the model-view-controller (MVC) pattern. We discussed this architectural pattern in section 5.2.1 above (see also figure 5-27). The MVC pattern provides a good approach to the design of any interactive system and in particular of systems where multiple user interfaces are an issue.

Forms-based interfaces

Forms are particularly important in business IS

Typical means for the interaction between a business information system and its users are forms and menus: Forms are the major mechanism to present and transfer information, and menus serve for navigation through the system. While menus are common in all types of software systems, forms are particularly important in business systems. For example, in an ERP (enterprise resource planning) system, all major objects – products, customers, suppliers, machines, warehouses etc. – are presented to the user via forms. New data are entered into forms, planning algorithms are started from forms, and data retrieved from the company's database are also shown in forms.

E-commerce occurs via forms

The same is true in *e-commerce*. Visiting an electronic shop on the Internet generally involves browsing through a product catalog that is presented with the help of forms. When the customer considers buying some items, these items are put into a shopping cart represented as a form. When he or she finally orders the articles, the order is shown as a form, as well as the order confirmation and the invoice. For this reason, forms deserve particular attention in the context of user interfaces of business information systems.

"Controls", "widgets"

Forms are composed of elements called "controls" (or "control elements"; sometimes also called "widgets"). A *control* is a GUI element through which a user and a form interact. Some controls are used to display information to the user; others let the user enter information or initiate an action, and some controls serve both purposes.

Controls are graphical elements with associated look and behavior. For example, a radio button is usually round, has a black dot in the center when clicked, and in a group of radio buttons, only one can be pressed at a time. Some user actions (such as clicking on a button or selecting an item from a text box) create an event that is to be handled by the program associated with the form (i.e. by an event handler – event-oriented programming).

Controls are usually available as predefined components from program libraries, packages or toolboxes. Software developers do not need to write the entire program code for them as they only include prewritten code into their modules.

GUI toolboxes

Integrated development environments (IDEs) such as Visual Studio, Eclipse and JBuilder provide toolboxes that support interface design with easy-to-use controls. Development productivity increases significantly if such a toolbox is available. In addition to this and other advantages, an IDE toolbox has a positive effect on the uniformity and consistency of the user interface. It not only gives the user interface a well-known look-and-feel (e.g. MS Windows like, Java Swing like), but the information system's forms and controls also look the same and behave in similar ways. Typical controls provided by a GUI toolbox include command and radio buttons, list boxes, picture boxes, text fields, labels, menus and much more. Figure 6-10 (chapter 6) shows a sample form containing some of these controls.

Accessibility

Nowadays, free access to information is considered valuable, sometimes even a human right. With regard to IT, accessibility describes the degree of ease with which it is possible for people to access an IT system (incl. IS). More specifically, accessibility refers to people with disabilities.

Accessibility demands that disabled persons should be enabled to access the same information as people without disabilities. Other aspects of accessibility comprise the support for elderly people and for people in "non-standard" situations:

Accessibility and "non-standard" situations

> "Users may not be able to see, hear, move, or may not be able to process some types of information easily or at all. They may have difficulty reading or comprehending text. They may not have or be able to use a keyboard or mouse. They may have a text-only screen, a small screen, or a slow Internet connection. They may not speak or understand fluently the language in which the document is written. They may be in a situation where their eyes, ears, or hands are busy or interfered with (e.g., driving to work, working in a loud environment, etc.)" [W3C 1999, p. 3].

National laws and regulations

Some countries have laws attempting to ensure that information technology is not a barrier for disabled persons. With regard to websites and web pages, the World Wide Web Consortium (W3C) in 1999 developed criteria and guidelines on how to make web content accessible to people with disabilities. These guidelines have been adopted into national laws and regulations by several countries. New W3C guidelines are under development, but they had not been finalized by the time this book went to press.

The W3C describes 14 accessibility guidelines which are shown in figure 5-41. Each of these guidelines is refined in many rules and practices, explaining in detail what is acceptable, what should be avoided, and how things should be realized. The W3C also defines detailed checkpoints, priorities (what "must", "should", and "may" be satisfied) and conformance levels depending on which checkpoints are satisfied.

While some guidelines are immediately understandable, others are described on a rather high abstraction level. The purpose of guidelines 1 and 2, for example, is obviously to help deaf, blind and color-blind people. The purpose of others can only be understood by reading the detailed rules and checkpoints. Why the need to "clarify natural language usage" (guideline 4), for example? The reason is that an automated tool reading the content of a web page to a blind person can pronounce words correctly only if it knows what language they are in. Suppose the default language is English and the screen reader does not know that the sentence contains French words. Pronouncing a phrase like "Skiing in France – downhill the Aiguille du Midi in Argentière" would sound rather funny, unless "Aiguille du Midi" and "Argentière" are marked up as being French.

Figure 5-41 W3C accessibility guidelines [W3C 1999]

No. Guideline	Explanation
1 Provide equivalent alternatives to auditory and visual content.	Provide content that, when presented to the user, conveys essentially the same function or purpose as auditory or visual content**.**
2 Don't rely on color alone.	Ensure that text and graphics are understandable when viewed without color.
3 Use markup and style sheets and do so properly.	Mark up documents with the proper structural elements. Control presentation with style sheets rather than with presentation elements and attributes.
4 Clarify natural language usage.	Use markup that facilitates pronunciation or interpretation of abbreviated or foreign text.
5 Create tables that transform gracefully.	Ensure that tables have necessary markup to be transformed by accessible browsers and other user agents.
6 Ensure that pages featuring new technologies transform gracefully.	Ensure that pages are accessible even when newer technologies are not supported or are turned off.
7 Ensure user control of time-sensitive content changes.	Ensure that moving, blinking, scrolling, or auto-updating objects or pages may be paused or stopped.
8 Ensure direct accessibility of embedded user interfaces.	Ensure that the user interface follows principles of accessible design: device-independent access to functionality, keyboard operability, self-voicing, etc.
9 Design for device independence.	Use features that enable activation of page elements via a variety of input devices.
10 Use interim solutions.	Use interim accessibility solutions so that assistive technologies and older browsers will operate correctly.
11 Use W3C technologies and guidelines.	Use W3C technologies (according to specification) and follow accessibility guidelines. Where it is not possible to use a W3C technology, or doing so results in material that does not transform gracefully, provide an alternative version of the content that is accessible.
12 Provide context and orientation information.	Provide context and orientation information to help users understand complex pages or elements.
13 Provide clear navigation mechanisms.	Provide clear and consistent navigation mechanisms -- orientation information, navigation bars, a site map, etc. -- to increase the likelihood that a person will find what they are looking for at a site.
14 Ensure that documents are clear and simple.	Ensure that documents are clear and simple so they may be more easily understood.

Automated tools can check accessibility

Website designers are encouraged, or forced by law, to follow the guidelines in order to make the website accessible. For example, if a web page contains pictures, the designer has to provide an alternate representation for each picture containing the same essential information as the picture itself (e.g. a textual description of the intent of the picture). Automated tools are available today to check whether a website is accessible or not.

Although the W3C accessibility guidelines are primarily guidelines for website design, the underlying ideas should be taken into consideration for any kind of information system that has an interface for human users. A color-blind person or a worker in a dark environment will have the same problems with any GUI, no matter whether it is a web front-end or a Java Swing front-end. Therefore we recommend keeping the W3C guidelines in mind when designing a user interface and following the rules as far as they are applicable.

User-interface prototyping

User-interface design is often an iterative process conducted with the help of prototyping. This is different from other design tasks. Conceptual work regarding architectural design and object design requires analytical skills, and results can be well documented in paper models (or in diagrams produced by a graphics tool). However, specifying a user interface in an abstract way seems to be difficult for most people.

Prototyping supports UI design

A more promising approach is to explore the needs of the user interface with the help of software prototypes, involving end-users and other stakeholders in the process. In this way, UI design is seen as an evolutionary process that starts with the development of the first prototype for requirements elicitation. We discussed this aspect in the section on requirements engineering (cf. section 5.1.2).

All process models and approaches discussed in chapter 4 include user-interface prototyping in one way or another. This is especially true for the iterative and non-standard approaches, i.e. RUP, agile development, XP (extreme programming) etc. The closer end-users are involved in the process, the more prototyping will be done automatically. In XP, for example, users are part of the development team. Continuous changes to the system and to the user interface in particular are quite likely to occur. Not even the waterfall model excludes user interface prototyping. A user-interface prototype can be developed within the requirements analysis and definition stage to help the analysts elicit the users' requirements (cf. section 4.2.1).

IDEs containing GUI toolboxes facilitate user-interface prototyping significantly. With simple drag-and-drop features and automatic layout features, it is very easy to create a graphical user interface on-the-fly, modify it and discard it if it is not deemed appropriate.

5.2.4 Designing the Database

Developing a relational data model

Designing a database means in the first place to develop a *relational data model*. This model is the foundation of the vast majority of today's databases and database management systems. Most corporate databases are built with the help of a relational database management system (RDBMS). Therefore, it is straightforward to model and store additional data for new IS in the same way.

However, the current favorite paradigm for software development is *object-orientation*. In the design process, classes and objects are modeled, resulting in class diagrams and object diagrams. Class definitions include data definitions (attributes), and the relationships between classes are also available. Due to this, the class diagrams appear to be a quite natural starting point to create a relational data model.

Object-oriented thinking is different from relational thinking

The problem is that object-oriented thinking is very different from thinking in relational concepts. Object-orientation is a paradigm for developing good-quality software systems composed of *objects* – where "quality" means properties such as maintainability, reliability, scalability, user-friendliness and usability. The relational model is a model for data entities and their relationships, based on mathematical set theory. Data are represented by relations that consist of *tuples*[§]. Operations on data are operations on sets.

"Object-relational impedance mismatch"

This gap has been called the *"object-relational impedance mismatch"* [Ambler 2003, p.105]. Overcoming this mismatch is an important task in the design stage. In this section we will address the major issues to be dealt with.

§ The term "relation" is not to be confused with the term "relationship" as used in class diagrams or entity-relationship diagrams. A relation in the relational data model is, according to mathematical set theory, a subset of the Cartesian product over the domains of the attributes.

Object-oriented database DBMS

A much simpler and more straightforward approach to creating a database for an object-oriented software system would be to use an *object-oriented database management system (ooDBMS)*. Such systems are capable of storing and managing "objects" – the same objects that are used in the programming language. This means that there is no need for the developer to bother with making relations (or tuples) out of classes (or objects). A number of object-oriented DBMSs are available on the market, but unfortunately these systems have not gained, and are unlikely to gain, significant market shares.

Object-relational DBMS

So-called *object-relational database management systems* are a step in between. They allow developers to use the data types (class names) of the programming language not only in their programs but also when accessing the database. The database is essentially a relational database that "understands" the application system's data types. Since most relational database management systems today provide this functionality, the term object-relational DBMS is not much used any more.

Relational DBMS

Using a relational database management system underneath an object-oriented software system requires two major steps to overcome the object-relational impedance mismatch:

1. Designing a persistence layer in the software system
2. Mapping classes and relationships to relations of the database

Designing a persistence layer

A relational database is normally accessed with the help of SQL (structured query language). Provided that developers know not only their programming language but also SQL, they could embed SQL statements directly in the program code. However, this approach is definitely not to be recommended because it means that the programmer has to do the conceptual work of mapping classes to relations and keep track of the details that implement the concepts each time data needs to be read from or written to the database.

A better approach is to decouple data access from the application program code and provide an object-oriented interface for data manipulation and definition. This could been done through separate data classes.

Persistence layer

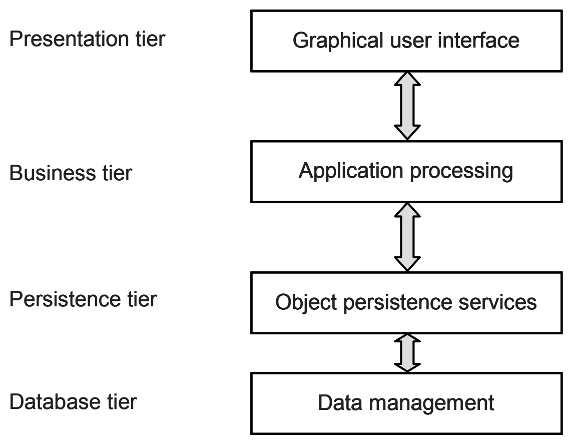
However, a more effective way is to design a so-called *persistence layer* in the software system that is independent of underlying implementations. On the one hand, this means that the application classes need to know the interface only and that they do not need to worry about the object-relational mapping. On the other hand, implementation changes do not affect the rest of the system. For example, the current

DBMS could be substituted by a new one, and this substitution will not have any impact on the information system as such (at least in theory).

In a layered architecture (cf. section 3.2.2, e.g. figure 3-3), the persistence layer can be seen as a part of the data tier or as a separate tier as shown in figure 5-42. In a service-oriented architecture (SOA, cf. section 3.3), persistence is a service that is available to software clients just like any other service.

A typical persistence layer (or service) contains *classes* which build SQL statements (such as select, insert, update), make objects persistent (i.e. to store objects), retrieve objects, handle collections of objects and manage transactions.

Figure 5-42 **Persistence tier in a four-tier architecture**



Persistence layer operations

Operations provided by a persistence layer include the following ones [Ambler 2005b, p. 8]:

– Build SQL statement
– Save object
– Retrieve object
– Delete object
– Get next object
– Get previous object

– Commit transaction
– Rollback transaction

Object-relational mapping

The second task in overcoming the object-relational impedance mismatch is to make relations out of the design model's classes and relationships. While the basic procedure of this mapping is straightforward and easy to follow, extra considerations are required in some special cases, complicating the process. The major parts of object-relational mapping comprise:

1) Mapping classes
2) Mapping generalization/specialization
3) Mapping associations

Making a relation for each class

Ad 1): *Mapping classes* is fairly simple but there are exceptions or special cases. The basic rule is to make a relation for each class, and to adopt the attributes of the class as attributes of the relation. (In database terminology, a relation is often called a *table*, an attribute is called a *column*, and a tuple is called a *row*, because of the tabular shape when the data are shown in a formatted form.)

Exceptions to the rule

Exceptions to the rule are:

a) Classes connected by generalization/specialization relationships are not always transformed in such a way that one relation corresponds to one class (see ad 2) below).
b) Some class attributes do not have a corresponding relational attribute. An example is a derived attribute whose value is computed from other values, such as an order total as the sum of all order items. While class attributes may represent derived values, attributes of relations normally don't.
c) Some class attributes may require more than one attribute in a relation. Relations have only scalar attributes while class attributes can be compound (i.e. the type of the attribute is a class). For example, an "address" attribute of a "Customer" class will require several columns in the database table, such as "street", "city", "zip code", "country" etc.

Note that the mappings have to be realized not only when objects are stored but in both directions! When a persistent object is referenced in the program, the data representing the object will be read from the database and the object will be reconstructed. For example, the order total attribute mentioned above may need to be recalculated from the individual order items, implying that order item data also have to be retrieved from the database.

Relational data models are often described in a notation used in relational algebra. Figure 5-43 shows in part a) an "Order" class connected with an "OrderItem" class as a UML design class diagram, as well as the corresponding relations in relational notation in part b). The name of a relation is written before the parentheses embracing the attributes of the relation. It is also common to prefix the name with an "R" (= relation) and a dot. So instead of defining a relation in the following way:

Order (<u>orderID</u>, customerID, orderDate, shippingDate, status, employeeID)

we would write:

R.Order (<u>orderID</u>, customerID, orderDate, shippingDate, status, employeeID)

Primary keys, foreign keys

The "orderID" attribute is underlined because it serves as the *primary key*. A primary key identifies a tuple in a relation uniquely. The other ID attributes ("customer ID" and "employeeID") are foreign keys. A foreign key is the primary key of another relation.

Data modeling profile

In order to express a data model in UML, class diagrams have to be used. UML does not provide a specific diagram for data modeling, but as an extension to UML, a so-called *data modeling profile* can be used. Unfortunately several authors have proposed different data modeling profiles, and the Object Management Group has not yet released a standard.

Data-modeling stereotypes

A data modeling profile, like UML profiles in general, contains a set of stereotypes explaining what a model element is used for. Stereotypes that have been proposed for data modeling include the following:

«Table»	– optional since all rectangles in a class diagram are tables
«PK»	– primary key
«FK»	– foreign key
«Subtype»	– inheritance relationship
«Composition»	– part-whole relationship
«Dependency»	– dependency relationship

A diagram using data modeling stereotypes, derived from the design class diagram of figure 5-43 a), is shown in part c) of the figure. In contrast to the design class diagram, the data diagram does not contain operations, and it shows more database-oriented information through the use of stereotypes.

Note that the following assumptions and transformations were made in the design class diagram and in its mappings:

– Business objects in the real world usually have identifiers following a strict classification or numbering scheme. We assume that such a scheme is given through an "IDType" class. This type is used for all important business objects. (It has been used, by the way, for the design classes in figure 5-34 before.) Therefore, the "Order" class has an attribute "orderID" of type "IDType".
– Where in the DCD a class name is used for the data type of an attribute (i.e. an abstract data type), this attribute is replaced by a foreign-key attribute pointing to the respective table that the class has been mapped to. For example, the types "Customer", "Employee" and "Product" are substituted by "IDType" foreign keys referencing the tables "Customer", "Employee" and "Product", respectively.
– The "Order" – "OrderItem" relationship is a composition, i.e. order items can only exist if an order exists. Therefore we did not give the design class "OrderItem" an "IDType" attribute. However, for the "OrderItem" database table a unique identifier is needed. Since the "productID" is not necessarily unique – an order might contain several suborders for the same product – we introduced an additional, unique identifier "itemID".

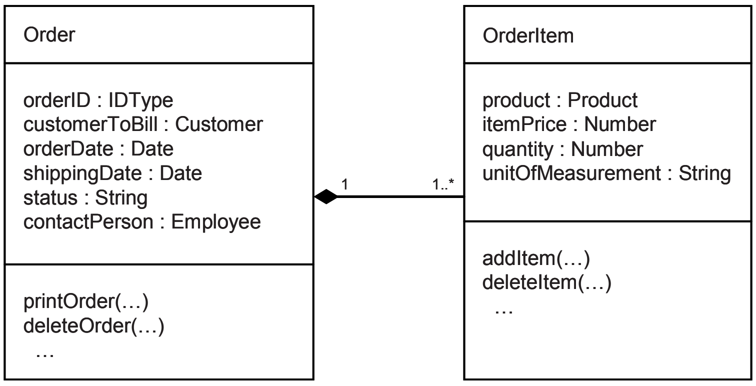
Mapping generalization/ specialization

Ad 2): *Mapping generalization/specialization* relationships is a special case of mapping classes. The question is whether the classes connected by generalization/specialization are all mapped to separate relations or not, and how this is done. Remember that specialization implies that the special classes have the same attributes as the general class, plus additional attributes (and additional operations which we do not consider here).

Proceeding according to the general rule – to make a relation out of each class – results in as many database tables as there are classes. The fact that these tables belong together is represented by a common primary-key attribute. The DBMS (or the application program) can then reconstruct a specialized object by joining the attributes of the child and parent objects.

Figure 5-43 Data modeling in UML

a) Design class diagram



b) Relations

Order (<u>orderID</u>, customerID, orderDate, shippingDate, status, employeeID)
Foreign keys: customerID, employeeID
OrderItem (<u>itemID</u>, productID, itemPrice, quantity, unitOfMeasurement)
Foreign key: productID

c) UML notation for data modeling

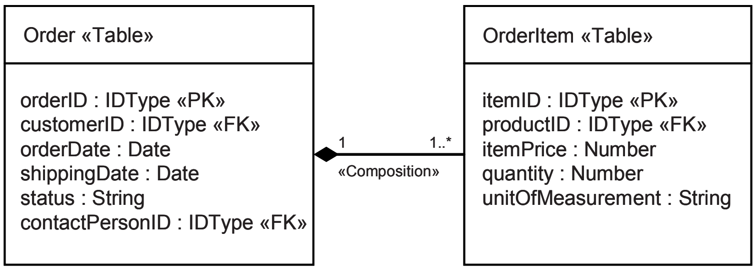
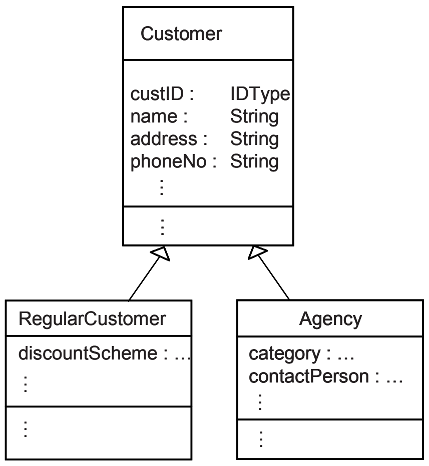


Figure 5-44 Specialization in classes and tables

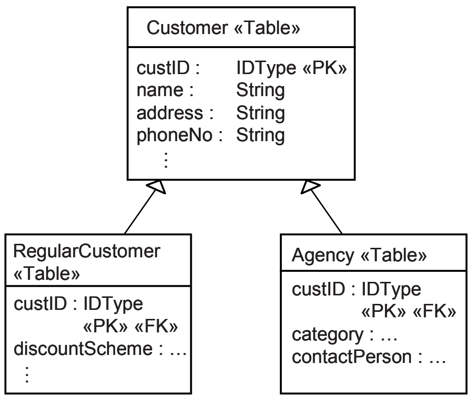
a) Design class diagram



b) Relations

Customer (<u>custID</u>, name, address, phoneNo, …)
RegularCustomer (<u>custID</u>, discountScheme)
Foreign key: custID
Agency (<u>custID</u>, category, contactPerson)
Foreign key: custID

c) UML notation for data modeling



Consider the specializations "RegularCustomer" and "Agency" of the "Customer" class in figure 5-44 a). Mapping these three classes to a relational data model yields three tables as follows:

Customer (<u>custID</u>, name, address, phoneNo, ...)
RegularCustomer (<u>custID</u>, discountScheme)
  Foreign key: custID
Agency (<u>custID</u>, category, contactPerson)
  Foreign key: custID

The UML equivalent is shown in part c) of the figure. This mapping approach is straightforward and compliant to proper data modeling principles. However, it has the drawback that two tables (or several tables, in the case of multi-level inheritance) in the database have to be joined each time a special customer object is accessed from the program.

Some approaches violate clean data modeling

Therefore other approaches have been proposed, but they tend to violate clean data modeling principles. One such proposal is to map all design classes to just one relation. Another one is to map only the special classes to database tables and duplicate the common attributes defined with the general class [e.g. Ambler 2003, pp. 233-234]. We do not recommend such practices and banish them into a footnote[§].

Mapping associations

Ad 3): Since relations and their attributes are the major concepts of a relational data model, they are also used to *map associations.* How this mapping is done depends on the multiplicities of the associations. We

§ a) In the case of mapping everything onto only one relation, this relation would comprise all attributes of all classes connected by generalization/specialization, plus one more attribute indicating what type of object it is. In the following relation, the "custType" attribute is used to distinguish which further attributes apply to a particular object:

Customer (<u>custID</u>, custType, name, address, phoneNo, discountScheme, category, contactPerson, ...)

However, using such a discriminator is a violation against normalization principles (the relation is not in the third normal form).

b) Mapping only the special classes to database tables and duplicating the common attributes creates redundancy in the database which can lead to inconsistencies later. Redundant attributes in our example are the attributes "name", "address" and "phoneNo" (originally of the "Customer" class) that are now adopted as attributes of both "RegularCustomer" and "Agency". Note that the two relations now have their own primary keys:

RegularCustomer (<u>regCustID</u>, name, address, phoneNo, discountScheme, ...)
Agency (<u>agencyID</u>, name, address, phoneNo, category, contactPerson, ...)

have to distinguish three cases: many-to-many, one-to-many, and one-to-one associations.

Many-to-many associations

A *many-to-many association* between two classes A and B has multiplicities of 1..* or 0..* at both ends of the association line. This means that any A object can be associated with many B objects, and any B object can be associated with many A objects.

Many-to-many associations are mapped via a connecting relation. This relation specifies through pairs "primary key A – primary key B" which tuple of A is connected with which tuple of B. In the design class model of figure 5-34, a many-to-many association existed between "Advertisement" and "Edition". A connecting relation should list which advertisement exactly (identified by an "adID") is published in which edition (identified by an "edID") of the paper.

In the DCD of figure 5-34 we already introduced an association class "Publication" that serves this purpose. The reason for modeling an association class was actually a different one – the association between "Advertisement" and "Edition" needed to have associations itself – but at the same time the association class "Publication" does exactly what we need now: providing pairs of "Advertisement" and "Edition" objects[§].

A connecting relation lists pairs of primary keys

This is exactly the purpose of a connecting relation, i.e. listing pairs of primary keys identifying tuples of the participating relations. At the same time, the two primary keys together constitute the primary key of the association class; i.e. this class has a compound primary key. This is why two attributes of the "Publication" relation are underlined:

Publication (<u>adID</u>, <u>edID</u>, state, ...)
  Foreign keys: adID, edID

One-to-many associations

In a *one-to-many association* between two classes A and B, the multiplicities are 1 or 0..1 at one end and 1..* or 0..* at the other end of the association line. This can be modeled in such a way that the relation with the "many" multiplicity at its end of the association line, say B, references a tuple of the other relation, say A, via a primary key of A. One of B's attributes is therefore a foreign key referencing a specific tuple of A. Looking again at figure 5-34, the association between "Publication" and "Customer" is a one-to-many association. Therefore, the "Publication" relation above is extended by another foreign-key attribute "customerID" referencing the customer who ordered publishing of the advertisement:

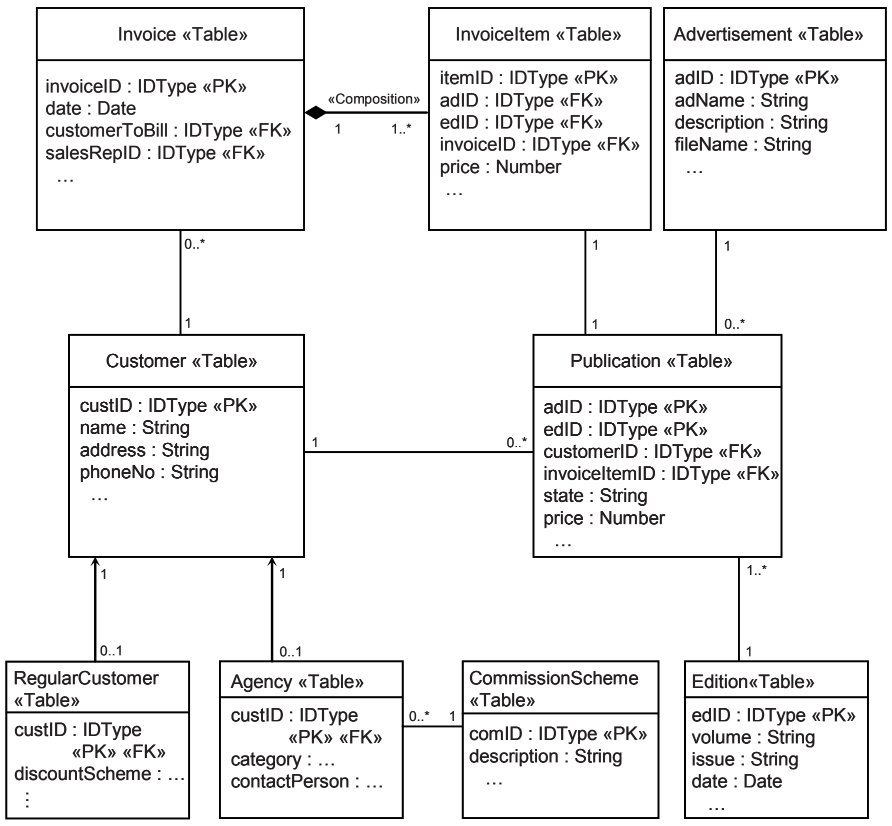
§ If we had not modeled the association class "Publication" before, we would now introduce a relation (e.g. "PublishedIn") connecting the "Advertisement" and "Edition" relations through pairs of primary-key values.

Publication (<u>adID</u>, <u>edID</u>, custID, status, ...)
Foreign keys: adID, edID, custID

One-to-one associations

In a *one-to-one association* between two classes A and B, the multiplicities are 1 or 0..1 at both ends of the association line. This means that exactly (or at most) one A object is associated with exactly (or at most) one B object. A common way of connecting the A and B relations is to include a reference to the A object in B and a reference to the B object in A, via foreign-key attributes.

Figure 5-45 **Data model as a class diagram using stereotypes**



In the example of figure 5-34, a one-to-one association was modeled between the classes "InvoiceItem" and "Publication". This situation can be mapped to the following relations, with additional attributes "invoiceItemID" and "adID, edID", respectively:

InvoiceItem (<u>itemID</u>, adID, edID, invoiceID, price, ...)
  Foreign key: adID, edID, invoiceID
Publication (<u>adID</u>, <u>edID</u>, custID, invoiceItemID, status, ...)
  Foreign keys: adID, edID, custID, invoiceItemID

Figure 5-45 summarizes the above mappings in a class diagram using UML data-modeling stereotypes. The diagram corresponds to the design class diagram in figure 5-34. Below are two aspects regarding multiplicities that have to be mentioned.

Firstly, note that the multiplicities of the association between "Advertisement" and "Edition" (0..* and 1..*) in the DCD had to be split up. The reason is that in the data model of figure 5-45 the relations "Advertisement" and "Edition" are not connected with each other but with the "Publication" relation. This relation has a one-to-many association (1 and 0..*) with "Advertisement" and another one-to-many association (1 and 0..*) with "Edition"!

The second point is how the generalization/specialization relationships between the "Customer" class and the classes "RegularCustomer" and "Agency" are modeled. Generalization/specialization is mapped to one-to-one relationships between the corresponding database tables, allowing also 0 occurrences (0..1 multiplicities). For example, 0..1 implies that if a customer is specialized into a regular customer, then the same customer cannot be specialized into an agency at the same time.

ER-relational mapping

ER modeling is widely used

Relational databases existed long before object-orientation became popular. In the pre-object-oriented times, relational databases were also created in a systematic way, but the starting point was often an entity-relationship model (ERM) and not a design class model. In fact, ER modeling is still a widely used technique for creating conceptual data models.

Same rules as for mapping a design class model

Mapping an entity-relationship model to a relational data model follows the same basic rules as discussed above for the mapping of a design class model. Actually these rules were established for ER-relational mapping a long time ago and later transferred to object-relational mapping. The three major steps corresponding to the ones explained for object-relational mapping are:

1. *Mapping entity types:* This is as straightforward as mapping classes. The general rule is to make a relation out of each entity type. Attributes of the relation correspond to the attributes of the entity type. If abstract data types or derived values have been used for entity attributes, then attribute mapping is not a one-to-one mapping as above. However, these cases are less common in ER modeling than in object-oriented modeling.
2. *Mapping generalization/specialization:* Generalization/specialization relationships are mapped in the same way as explained above. Each entity type is represented by one relation. The relations for the specialized entity types are defined with the same primary keys as the relation for the general entity type.
3. *Mapping other relationships* (than generalization/specialization relationships): This step is analogous to mapping associations between classes. Many-to-many relationships are transformed using connecting relations. A one-to-many relationship between two entity types A and B is mapped with the help of a foreign-key attribute in B which references a tuple in A. In a one-to-one relationship, a tuple of A points to the associated tuple of B via a foreign key and vice versa.

Mapping the ER model of figure 5-25

To illustrate ER-relational mapping through an example, take another look at figure 5-25. In this figure, an entity-relationship model was created for the same issues as modeled in the design class diagram of figure 5-34. Going through the steps 1) to 3) will yield exactly the same relational data model that was created by object-relational mapping. The result can be studied in figure 5-45.

Note that in the ERM an "Invoice item" was modeled as a one-to-many relationship between the entity types "Invoice" and "Publication". This relationship was not mapped through a foreign-key attribute but through a separate relation. The reason is that the "Invoice item" relationship in the ER model had *attributes.* In order to represent attributes in a relational data model, a relation is needed.

XML databases

We have assumed so far that we are designing a relational database. This is a reasonable assumption because most real-world databases today are based on the relational data model. On the other hand, XML has gained wide acceptance in many areas of software engineering, in particular in Internet computing.

Service orientation as a new paradigm for software development has brought the concept of software as a service. A service is a software module provided over a network. A service-oriented architecture (SOA) as discussed in section 3.3 comprises services, e.g. web services or enterprise services. Web services are usually invoked via the SOAP protocol over an intranet or the Internet, and SOAP is based on XML.

Database requests and responses are sent as XML messages

Let us assume that a service invocation is about retrieving information stored in the company's database. A common practice today is that the database server is a node in the company's intranet which provides database services. Therefore, the database request is sent as an XML message over the network. The response containing the information retrieved from the database also comes back as an XML message. How does this happen? The answer is that a piece of software in between wraps the actual database information into XML format.

To get an impression of what this looks like, revisit the figures 3-7 and 3-8 in chapter 3. Figure 3-7 contains a service request in XML – namely to provide information about the product with ID "A-1088". The values returned are "racing bike" (name), "low-end racing bike for upward mobile professionals" (description), "230.99" (price), and "13" (quantityAvailable). Most likely these values were retrieved from a product database with an SQL statement such as:

```
Select name, description, price, quantityAvailable
from Product
where productID = "A-1088"
```

Obviously the XML request in the SOAP envelope was mapped onto such an SQL command. Likewise the tuple of values returned was wrapped into an XML response such as the one in figure 3-8. Although the example in figures 3-7 and 3-8 was actually about invocation of a higher-level web service for data management ("MasterDataService") and not about immediate database access, similar XML code would be needed to send a request to a database service and to receive the response.

This example makes clear that it would be nice if the DBMS understood XML directly, instead of forcing the developer to make SQL calls (or, as in a more software-engineering way, calls to persistence-layer operations) out of XML messages, be it manually or with the help of a middleware; and vice versa.

Types of XML databases

This is the motivation for *XML databases*. The XML:DB Initiative, an industry consortium promoting XML databases, identifies three major types – native, XML enabled and hybrid databases. They are characterized as follows [XML:DB 2003]:

– A *native XML database (NXD):* a) defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model; b) has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage; c) does not require any particular underlying physical storage model. For example, it can be built on a relational, hierarchical or object-oriented database, or use a proprietary storage format such as indexed, compressed files.
– An *XML enabled database (XEDB)* is a database that has an added XML mapping layer provided either by the database vendor or a third party. This mapping layer manages the storage and retrieval of XML data. Data that is mapped onto the database is mapped onto specific formats of the DBMS vendor, and the original XML metadata and structure may be lost. Data retrieved as XML is *not* guaranteed to have originated in XML form.
– A *hybrid XML database (HXD)* is a database that can be treated as either a native XML database or as an XML enabled database.

The major commercial database management systems today are XML enabled. They accept input written in XML and render output in XML, but their internal structure is relational. This means that they map XML elements onto relations, tuples and attributes, and vice versa.

5.2.5 Other Approaches to Design: SD/CD

DFDs describe activities and data flows, not classes

For more than two decades, SA (structured analysis) was the dominating approach regarding systems analysis and in particular, requirements engineering. When SA is applied, the results of the requirements stage are data flow diagrams (DFDs) and a data dictionary, possibly supplemented by decision trees, decision tables and procedural descriptions in so-called "structured English". In contrast to UML diagrams, DFDs describe *activities* connected by *data flows* (not classes or objects). Data are passed from one activity to the next.

Structured design/composite design)

Several approaches to create a design from DFDs, as a follow-up to SA, were developed in the 1970s by well-known SA authors such as

Larry Constantine, Ed Yourdon, Glen Myers and Wayne Stevens [Stevens 1974, Yourdon 1979]. They were called *SD (structured design)* and *CD (composite design),* often together referred to as *SD/CD*.

Module cohesion and coupling

The main ideas are attributed to Larry Contantine and his work on module cohesion and module coupling. These principles are still considered important characteristics of modern software quality. A major goal of his design approach was to achieve loose coupling between and tight cohesion within modules.

SD/CD is still a popular design methodology wherever SA is the favorite approach to requirements engineering. The primary question answered by SD/CD is: How can a design (in terms of software modules) be derived from the requirements (described as DFDs)? The DFDs describe how data flow between activities. However, what is needed now are software components and their interrelationships (e.g. which module calls which other module) – or in other words, a decomposition of the entire software system into modules.

Structure chart: decomposing the system into modules

The major outcome of SD/CD is a *structure chart*. In this sense the aim of SD/CD is to create a structure chart from the data flow diagrams. A structure chart is a tree diagram showing the hierarchy of modules and their relationships [Martin 1986, p. 181-190].

Several approaches have been proposed to derive a structure chart from a DFD. The two most common ones are: transform analysis (for data-oriented systems) and transaction analysis (for transaction-oriented, e.g. interactive systems).

Transform analysis

Transform analysis is used for problems where the classical data processing aspect is dominating, i.e. the system is mainly to process input data and to create output data:

Input → processing → output

Dividing a level-1 DFD into three parts

While this scheme looks quite simple, each of the three parts can be rather complex. Therefore the first step of transform analysis is to divide a level-1 data flow diagram into three parts called:

- *Afferent branch* – contains those DFD processes that are responsible for reading input data and transforming this data into such a shape that the essential task of the system can be carried out subsequently.
- *Central transform* – concerned only with the logical processing and not with input or output related considerations.

– *Efferent branch* – consists of those processes which transform internal data into an output form suitable for the user.

There may be more than one afferent branch, more than one central transform, and more than one efferent branch in a structure chart.

To illustrate transform analysis, consider the data flow diagram of figure 5-46 for a newspaper subscription system. It is split up into an afferent branch, a central transform and an efferent branch.

Figure 5-46 Example of a data flow diagram divided into three parts

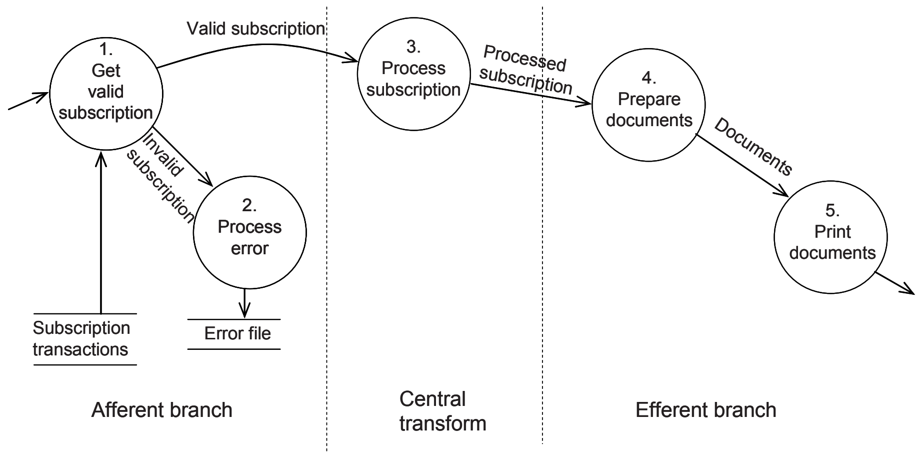


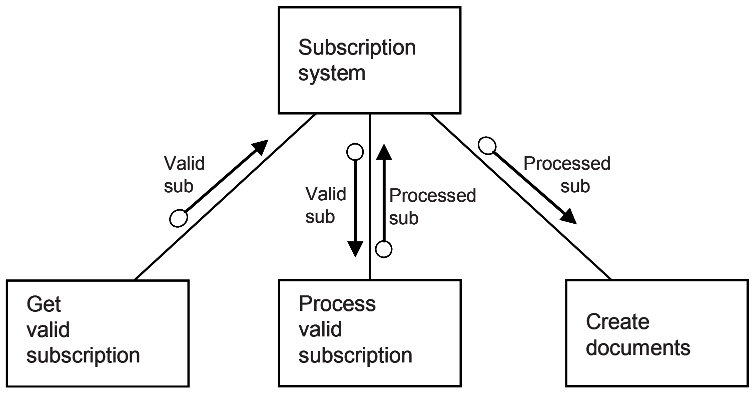
Figure 5-47 shows the top-level structure chart which corresponds to the DFD of figure 5-46. It consists of three top-level modules: "Get valid subscription", "Process valid subscription" and "Create documents". Arrows with hollow circles indicate data passed from one module to another. For example, a "Valid sub" (valid subscription) is passed from the main module to the module "Process valid subscription", and a "Processed sub" is passed back.

In the subsequent steps, called factoring, the top-level structure chart is refined and extended. This means that the modules representing the central transform and the input and output branches are refined into submodules. The factoring process may continue several levels down, de-

Factoring

pending on the complexity and size of the system. Additional modules can be included, e.g. from program libraries and for error-handling.

Figure 5-47 High-level structure chart with a central transform



Transaction analysis

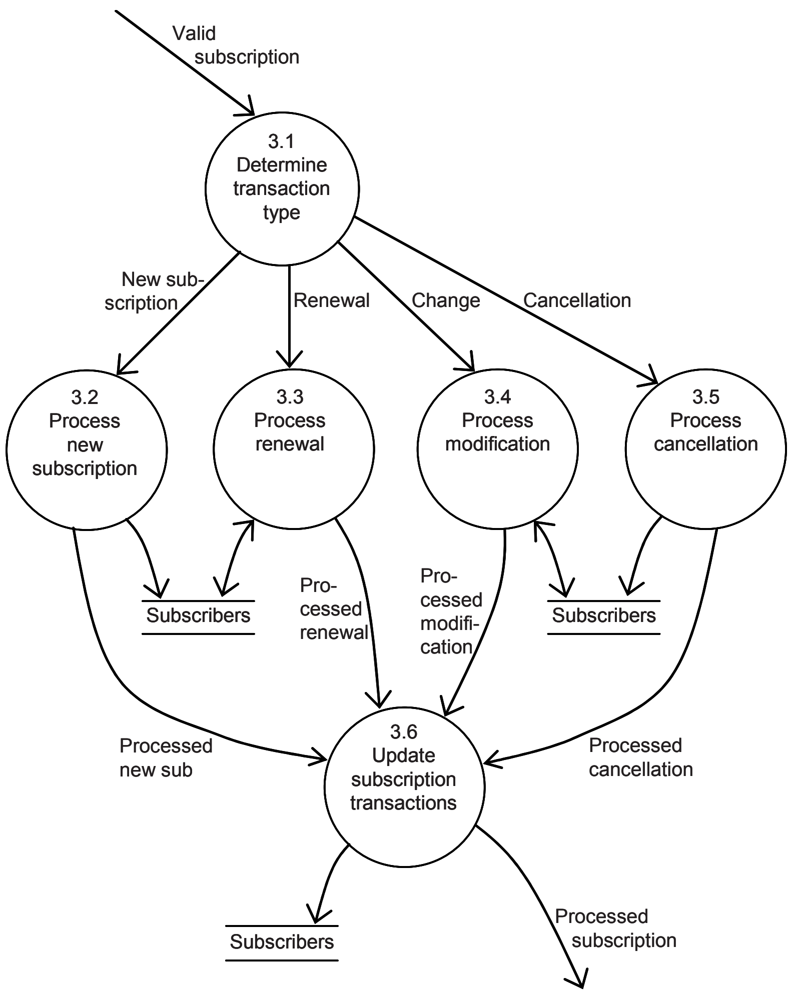
The second approach, transaction analysis, is centered around the notion of a transaction. In SD/CD, a transaction is an action triggered by some data element, in particular when that data element receives its value through user input.

Transaction center

A transaction center is a module that distinguishes between different types of processing depending on the mentioned data element (or more broadly speaking, on the desired type of action). The respective modules invoked by the transaction center are called transaction modules. A typical transaction center is an activity from which a number of alternative data flows originate. However, only one of several succeeding processes will be performed.

Often the transaction centers are easy to spot in a data flow diagram. In the DFD refining the "Process subscription" node of figure 5-46, such an activity – with multiple mutually exclusive outgoing data flows – is "3.1 Determine transaction type". This is illustrated in figure 5-48. Depending on the outcome of the "Determine transaction type" activity, either process 3.2, 3.3, 3.4 or 3.5, but only one of these processes, will be performed.

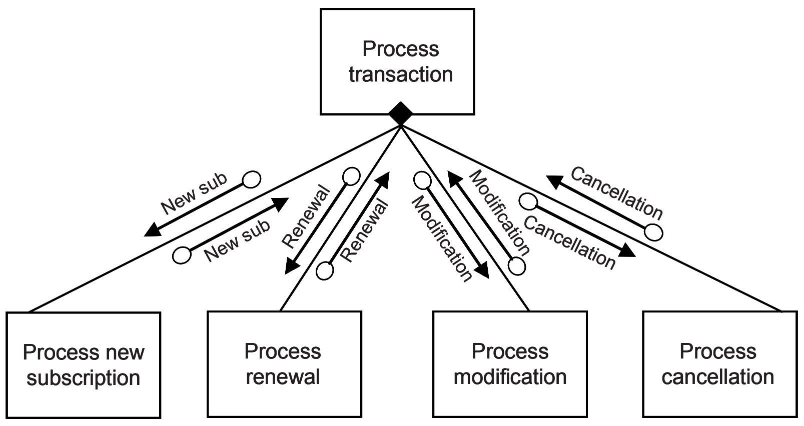
Figure 5-48 Example of a DFD with alternative data flows



Note that the datastore "Subscribers" is shown three times in the diagram although it exists only once. The main reason for this is to avoid crossing lines. Since the processes 3.2 to 3.5 are all connected with the datastore, at least two lines would intersect with others if the datastore were put in the diagram only once[§].

In a structure chart, a transaction center is a module that has several child modules, but only one will be invoked. The transaction center is marked by a black diamond from which the connections to the subordinate modules originate. Figure 5-49 shows, as an example, a transaction center corresponding to the "Determine transaction type" DFD node, and transaction modules corresponding to the DFD processes 3.2 to 3.5.

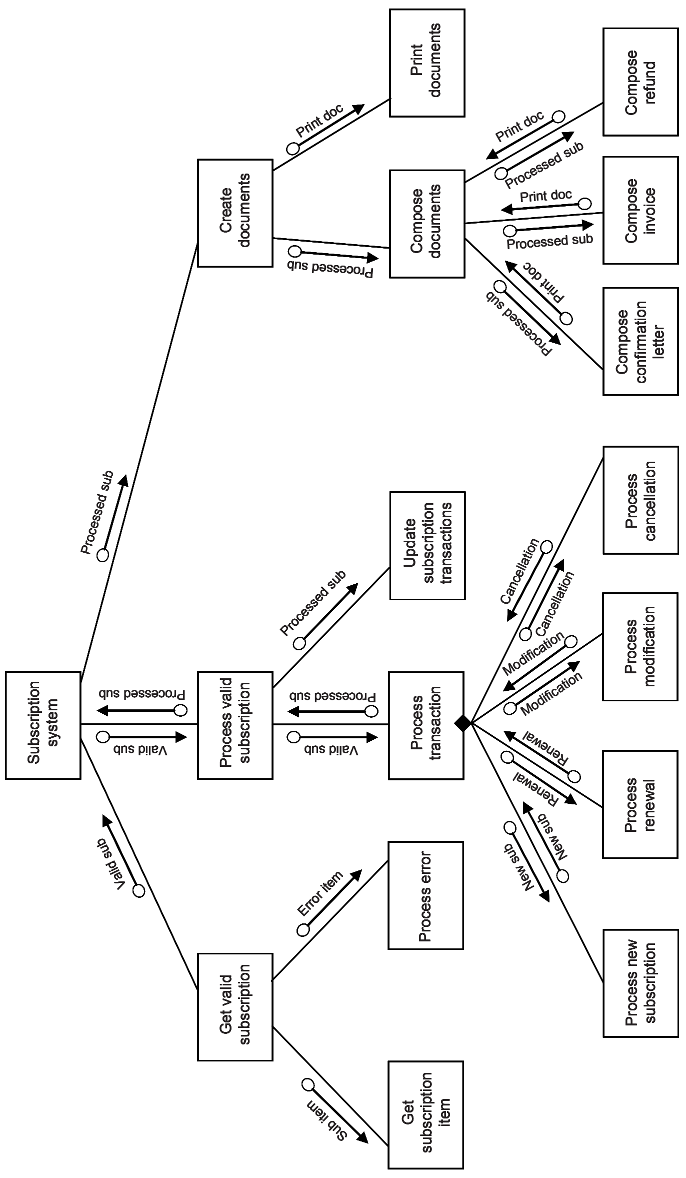
Figure 5-49 Structure chart with a transaction center



Transform analysis and transaction analysis are not mutually exclusive approaches. In the design of many non-trivial systems, both central transforms and transaction centers can be found. The complete structure chart for the subscription system, composed of the subdiagrams above, also contains both a central transform part ("Process valid subscription" and submodules) and a transaction center ("Process transaction"). This structure chart is shown in Figure 5-50.

§ Duplication of symbols is the preferred way of avoiding line crossing in data flow diagrams. In the Gane-Sarson notation [Gane 1979], a special marker is used to indicate duplicates, whereas in the original DeMarco notation [DeMarco 1979] the symbols are just shown more than once.

Figure 5-50 Structure chart for the subscription system



The flow of control, beyond invocation of subordinate modules, is normally not shown in a structure chart. However, since modules may be called repeatedly (iteration) or conditionally (selection), some authors have proposed symbols and conventions for control structures such as:

– Iteration – indicated by a curved arrow (↺) around the top ends of the lines connecting modules.
– Selection – indicated by a diamond (◇), similar to the black diamond used in a transaction center.
– Sequence – placing modules from left to right implies sequential execution of the modules in this order.

Subsequent SD/CD steps

While the main task in SD/CD is to create a structure chart from DFDs, the design methodology comprises more steps. In particular, the design has to be evaluated according to the cohesion and coupling criteria. The design may be modified in order to improve module cohesion and loosen module couplings. Afterwards, the design is prepared for implementation, which means primarily that the programs or program units into which the design modules will be split up or combined are specified.

SD/CD is suited for data-oriented problems

SD/CD is a typical top-down design methodology for data-oriented problems, following the principle of functional decomposition. Deriving high-level system structures from DFDs is quite straightforward. However, this is only the starting point. Most of the work is still left to the so-called *factoring* (i.e. further decomposition). This is a top-down process that can be as difficult as with any other methodology.

While SD/CD was very popular in the 1980s and 1990s, it has lost much of its importance due to the rise of object-oriented analysis and design. Classes and objects do not translate easily into procedural modules. However, since many legacy systems were built according to SD/CD, knowing this approach is helpful, e.g. in reengineering projects.

5.3 Upper CASE

Much of the analysis and design work consists of creating and documenting models in diagrams. All UML diagrams and all other diagrams described above must be drawn! Not only is the creation of diagrams a lot of work, but changing them later in a consistent way is even worse. Doing this work on paper can be a nightmare.

Simple drawing features in Office programs such as Word and Powerpoint help a little. Specific drawing tools such as Visio and Corel Draw are somewhat better because they adjust elements automatically. However, they do not "understand" what a drawing is about, and therefore cannot check whether it is obviously wrong or not. Partial understanding of the semantics of the underlying models would be helpful in many situations.

5.3.1 Automating Diagrams

Propagating model changes

Suppose a design model consists of several diagrams, for example design class diagrams, sequence diagrams and activity diagrams. Let us assume that the designer wishes to delete a class in a DCD, enter a new one into the design model, and change the name of a third one. The DCD is affected, and it is likely that the other diagrams are too. Obviously there cannot be any more activities and interactions involving the deleted class. On the other hand, additional activities and interactions with the new class have to be considered. Changing the name of the third class is fairly simple, but still it needs to be done in every diagram where that class is used.

Doing these modifications on paper models means a lot of eraser, scissors and glue usage. Wouldn't it be nice if the graphical tool "knew" which diagrams are affected, where to find the classes, relationships and other connections that need to be redefined, and what to tell the designer about the necessary changes? In order to be able to do all the required checking, the tool must possess a certain level of understanding about the meaning of the diagrams.

CASE tools

Graphical tools with such capabilities have been around since the late 1980s and early 1990s. They are called CASE tools. CASE stands for "computer aided software engineering" – an analogy to other CA-terms such as CAD (computer aided design) and CAP (computer aided planning) used in the manufacturing industry.

CASE tools are very comprehensive software systems. The more stages of the software life cycle a CASE tools supports, the more diagram types it has to offer. A complete UML CASE tool has to support 13 types of diagrams in a consistent way, not counting numerous UML extensions such as profiles.

What makes CASE tools especially interesting for a development project is the integration of different tools and in particular, integration of the results (diagrams) produced with the tools. This means three things:

1. A diagram created in one SLC stage can be used in the next stage as input for its available tools.
2. Transformation of one diagrammatic representation of the same thing into another representation can be automated to a certain degree.
3. Transformation of models into code can also be automated to a certain degree (code generation).

Repository

An enabler for the integration of tools and for automated transformations between the tools is a common *repository* in which all information and meta-information is stored. This means in particular that all results created during analysis, design and coding activities – diagrams, class definitions, program code etc. – are available in electronic form in the repository. Various types of visual representations can then be generated from the information stored in the repository.

Comprehensive CASE tools supporting the work throughout all SLC stages with the same quality were promised by CASE advocates but never actually delivered. Some tools were good at modeling, in particular for analysis and design, but not at generating code. Others produced good code or code templates from models, but were weak regarding analysis and design.

Upper and lower CASE

This situation led in the 1990s to the distinction between "upper CASE" and "lower CASE" tools. "Upper" and "lower" actually refer to the positions of the respective activities in a SLC model such as the waterfall model (cf. figure 4-2):

– *Upper CASE* means tool support for the early stages of the software life cycle, especially for analysis (requirements engineering) and design tasks.
– *Lower CASE* means tool support for stages further down the software life cycle: program design, coding, testing and debugging.
– *I-CASE (integrated CASE)* means tool support for all stages in such a way that tools are integrated [Martin 1989, p. 54].

Integrated CASE

I-CASE is the term for what was actually desired: high-quality CASE support for all software life cycle activities. This would mean: integrated tools across all stages (upper and lower) of the software life cycle, including integrated results created with the tools. This would finally result in code that is automatically generated from the models.

I-CASE was a lofty goal at the time. Even though some very powerful CASE tools existed – examples are ADW (Application Development Workbench) and IEF (Information Engineering Facility) [Stone 1993] – they were not equally powerful for all SLC stages, from analysis all the way down to code generation. While analysis and design models were supported by convenient modeling tools, code generation was cumbersome and error-prone.

IBM's "repository manager"

Another problem was the creation of an *open repository* with interfaces to plug in tools from different vendors. One of the biggest software-project failures reported in history was actually the famous "repository manager" project by IBM in the 1980s and 1990s [Sagawa 1990]. It failed because the task of building such a comprehensive repository was just too complex at the time, even for an IT heavyweight such as IBM.

Today, a new generation of CASE tools are available, although they are not always explicitly called CASE tools. Outstanding features of the early CASE tools were the use of graphics and visualization of information. Nowadays most tools targeted at human users are graphical tools, because visualizing information is considered user-friendly. Some of today's toolsets have names including the term "studio", in particular lower CASE tools (e.g. Visual Studio, WebSphere Studio).

Today's upper CASE tools

We will continue to use the term CASE because it is tool and vendor independent. Since analysis and design is the topic of this chapter, the focus is on upper CASE tools. These tools support typical analysis and design activities, usually with the help of UML diagrams. Well-known tools include the following:

– Software Architect (full name: IBM Rational Software Architect§)
– Software Modeler (full name: IBM Rational Software Modeler)
– Rational Rose (full name: IBM Rational Rose#)

§ Software Architect, Software Modeler and Rational Rose are available from IBM Rational (http://www-306.ibm.com/software/rational/).

# Renamed after the acquisition of Rational Corp. by IBM in 2003. Rational Rose is a comprehensive family of UML modeling and development tools, one of the most widely used UML toolsets. However, it supports only UML up to version 1.4.

- Enterprise Architect (by Sparx Systems, Australia; http://www.sparxsystems.com/products/ea.html)
- Together (full name: Borland Together; http://www.borland.com/us/products/together/)
- objectiF (by MicroTool, Germany; http://www.microtool.de/objectiF/en/)
- Innovator (by MID, Germany; http://www.mid.de)

5.3.2 An Example: Modeling with a CASE Tool

In this section, we will look at a sequence of diagrams created with an upper CASE tool. The example is about an advertisement ordering system offered by a newspaper publishing company. It includes some diagrams that were manually drawn earlier in this chapter. The modeling tool used to create the diagrams is *Enterprise Architect*.

Main actors

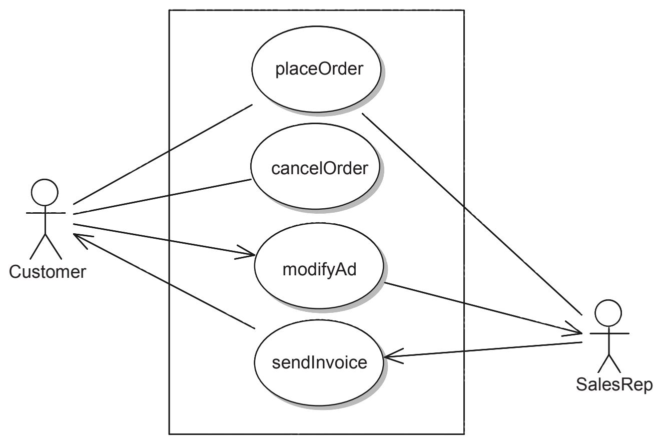
Figure 5-51 illustrates the main actors using the system. *Customers* can place and cancel orders. An order means that one or more advertisements are to be published in a particular edition of the paper. Since advertisements are often reused, the publisher keeps them stored and offers to do simple changes inhouse (e.g. updating a text field such as application deadline).

Customers appreciate this service because they do not have to pay their own advertising agency to produce a new advertisement file. Therefore a use case "modifyAd" is modeled, expressing that this customer initiates a modification in the stored advertisement and the *sales representative* responsible for the customer will see that the change is done. The sales representative also invoices the customer later.

Domain concepts

The *domain concepts* relevant for requirements engineering are shown in figure 5-52. Advertisements are associated with particular newspaper editions in which they are supposed to appear. Customers order the publication of an advertisement for a particular edition. An invoice can consist of several invoice items, each of which refers to a published advertisement.

Figure 5-51 **Use-case diagram for advertisement-ordering system**



Design class diagram

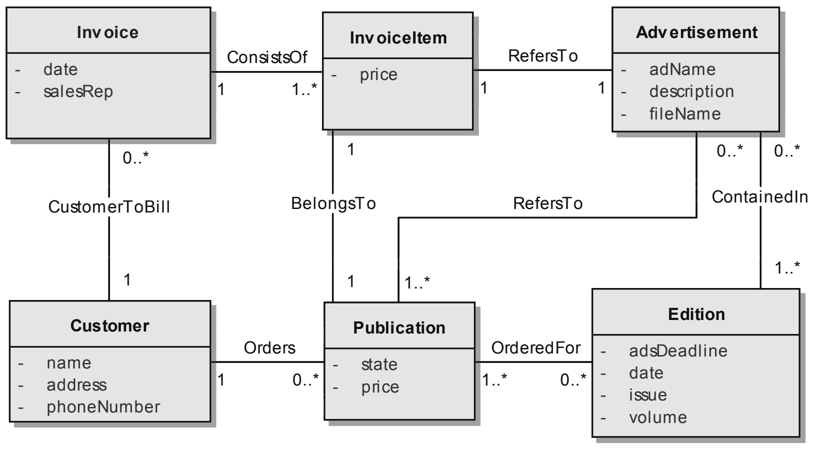
A *design class diagram* containing some classes derived from the domain model and additional classes created in the design process is shown in figure 5-53. It is basically the same diagram as the one drawn in figure 5-34 before, yet in a tool-specific notation. Private class members are indicated by a hyphen whereas public members have a plus sign. The notation for methods is Java-like. Methods can return a value. For example, the "getTotal" method returns a numerical value of type "double", and the "adFileExists" method returns "true" or "false". Underlined methods are static methods (class methods).

The "Customer" class was specialized in the design process into an "Agency" class and a "RegularCustomer" class. The reason is that the two types of customers are treated differently. Furthermore, the "CommissionScheme" for agencies is quite differentiated. Therefore it was modeled as a separate class. The "Customer" class has two important methods:

1. The "placeOrder" method is for new advertisements. When a customer places an order for a new advertisement, the file containing it plus the volume and the issue in which it is to be published have to be provided as parameters.

2. The "placeRepeatOrder" method should be used for existing advertisements, i.e. advertisements that have been published before. The first parameter here ("oldAd") is a reference to the advertisement in the advertisement-ordering system.

Figure 5-52 **Domain model for advertisement-ordering system**

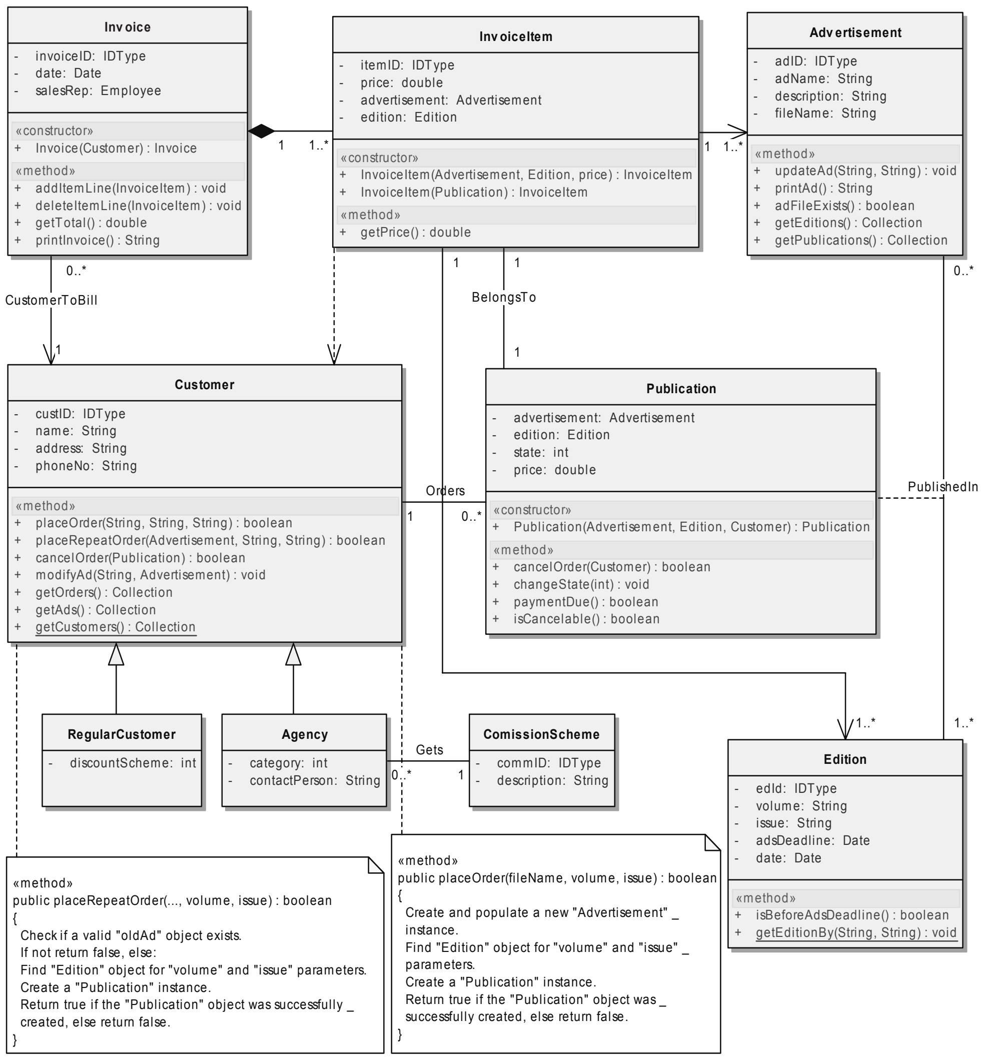


The relationships between advertisement objects, editions of the newspaper and the publication of an advertisement in a particular edition was reconsidered and obviously modeled in a different way than in the initial domain model. Remember that the purpose of a domain model is to capture the essential concepts of the application domain without giving too much thought about details.

Advertisements, editions and publication had already been considered in domain modeling as conceptual classes. However, in the design it showed that a "Publication" is actually something that associates an "Advertisement" object with an "Edition" object. Therefore it was modeled as an association class.

The DCD created by the CASE tool looks quite similar to the manually created diagram shown in figure 5-34. The generated notation is more or less the same as described for class diagrams in the official UML reference.

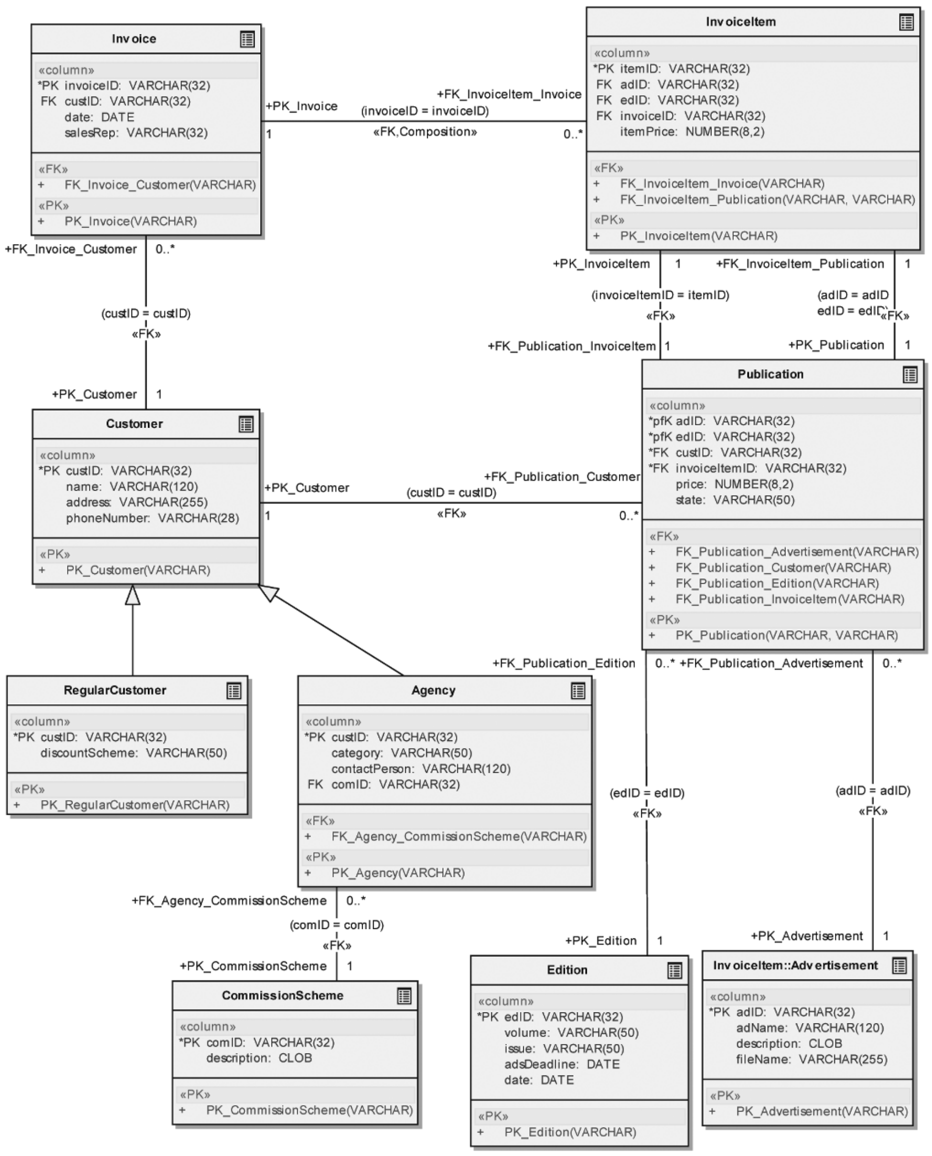
Figure 5-53 **Design class model for advertisement-ordering system**


Invoice
- invoiceID: IDType
- date: Date
- salesRep: Employee
«constructor»
+ Invoice(Customer) : Invoice
«method»
+ addItemLine(InvoiceItem) : void
+ deleteItemLine(InvoiceItem) : void
+ getTotal() : double
+ printInvoice() : String
InvoiceItem
- itemID: IDType
- price: double
- advertisement: Advertisement
- edition: Edition
«constructor»
+ InvoiceItem(Advertisement, Edition, price) : InvoiceItem
+ InvoiceItem(Publication) : InvoiceItem
«method»
+ getPrice() : double
Advertisement
- adID: IDType
- adName: String
- description: String
- fileName: String
«method»
+ updateAd(String, String) : void
+ printAd() : String
+ adFileExists() : boolean
+ getEditions() : Collection
+ getPublications() : Collection
1
1..*
1
1..*
0..*
CustomerToBill
1
1
1
BelongsTo
1
0..*
Customer
- custID: IDType
- name: String
- address: String
- phoneNo: String
«method»
+ placeOrder(String, String, String) : boolean
+ placeRepeatOrder(Advertisement, String, String) : boolean
+ cancelOrder(Publication) : boolean
+ modifyAd(String, Advertisement) : void
+ getOrders() : Collection
+ getAds() : Collection
+ getCustomers() : Collection
Orders
1
0..*
Publication
- advertisement: Advertisement
- edition: Edition
- state: int
- price: double
«constructor»
+ Publication(Advertisement, Edition, Customer) : Publication
«method»
+ cancelOrder(Customer) : boolean
+ changeState(int) : void
+ paymentDue() : boolean
+ isCancelable() : boolean
PublishedIn
RegularCustomer
- discountScheme: int
Agency
- category: int
- contactPerson: String
Gets
0..*
1
ComissionScheme
- commID: IDType
- description: String
1..*
1..*
Edition
- edId: IDType
- volume: String
- issue: String
- adsDeadline: Date
- date: Date
«method»
+ isBeforeAdsDeadline() : boolean
+ getEditionBy(String, String) : void

```
«method»
public placeRepeatOrder(..., volume, issue) : boolean
{
  Check if a valid "oldAd" object exists.
  If not return false, else:
  Find "Edition" object for "volume" and "issue" parameters.
  Create a "Publication" instance.
  Return true if the "Publication" object was successfully _
  created, else return false.
}
```

```
«method»
public placeOrder(fileName, volume, issue) : boolean
{
  Create and populate a new "Advertisement" _
  instance.
  Find "Edition" object for "volume" and "issue" _
  parameters.
  Create a "Publication" instance.
  Return true if the "Publication" object was _
  successfully created, else return false.
}
```

Figure 5-54 **Data model for advertisement-ordering system**



Many implementation-specific details were generated

Unfortunately this is not the case in the next step: deriving a UML data model from the design class diagram. In this step, the particular CASE tool we used generated plenty of implementation-specific details that we actually would not need at this stage of the process where we just want to create a plain data model.

Generated data model

The generated *data model* is shown in figure 5-54. It contains as many tables as there were classes in the design class diagram. Each table corresponds to a class of the DCD. Instead of the UML stereotype «Table», the class name compartment contains a little table icon on the right-hand side.

Note that the attributes and the data types were not mapped exactly. The reason is that our CASE tool declares attributes only in a database-dependent way. The user has to specify a concrete DBMS first. Thus the data types in the diagram are data types supported by a concrete DBMS (we chose Oracle in order to be able to continue). The "String" type used in the design classes, for example, is now an Oracle "varchar" type of a specified length.

Limitations by the CASE tool

The CASE tool we used imposes more severe limitations. A class name cannot be used as a data type of the primary key. Therefore, the "IDType" class had to be replaced by one of the supported types. We chose a "varchar(32)" type, i.e. a string 32 characters long, to map key attributes (assuming that a key can be represented as a sequence of characters and subdivided into substrings with appropriate meanings).

The operations compartments contain generated methods which are actually methods ensuring primary and foreign key constraints. It is questionable to show such implementation-oriented details in a data model. However, the tool does this (and others also do), so we have to cope with it. Likewise, the joining conditions written on the association lines are very much a matter of implementation (how are data retrieved from different tables in SQL) and not of modeling.

Indicating foreign-key relationships

A useful piece of information indicating which class a foreign key actually points to is generated by the tool and written as a role on the association line. For example, foreign-key relationships exist between "Customer" and "Invoice", and between "InvoiceItem" and "Invoice". "+FK_Invoice_Customer" and "+FK_"InvoiceItem_Invoice" on the association lines indicate these roles. The "+FK_"InvoiceItem_Invoice" role indicates that a foreign key points from the "InvoiceItem" to the "Invoice" table. The stereotype «Composition» provides in addition the information that an "Invoice" objects consists of "InvoiceItem" objects.

5.3.3 On to Implementation

The design class diagram and the data model we created with the CASE tool are the two major models that we will use as input in the next process stage: implementation. Of course, more UML diagrams can be – and in non-trivial projects usually are – created in the analysis and design activities, including state machine, activity, sequence, communication and package diagrams. The upper CASE tool we have been employing – just as the other tools mentioned in section 5.3.1 – supports the creation of these diagram types, too.

Comments can be used to describe procedural logic

Methods which are not straightforward to implement (e.g. methods containing complicated algorithms) may be outlined in the design stage in pseudo-code or in a programming language. This facilitates the coding in that the programmers do not need to rethink the same problems the designers already solved. In UML, such explanations are added with the help of a comment. As an example, the "placeOrder" and "placeRepeatOrder" methods in figure 5-53 are supplemented by comments describing the procedural logic.

Design class models and data models are the most import ones

The two mentioned models (design class model and data model) are the most important ones of all in that they specify the programs and the database tables that need to be implemented. From a CASE perspective, these models are the ones that today's CASE tools handle in a sufficiently reliable way: The tools allow the developer to generate adequate code, i.e. program code in a language such as Java, and database code in a DDL (database definition language). This will be illustrated in more detail in sections 6.1.1 and 6.1.2 below.

Code generation does not replace programmers but relieves them of some schematic work, such as writing class headers, attributes, methods' signatures etc. These things can be derived automatically from the specifications in a design class diagram. Code for things that have not been specified (e.g. how exactly will a method do its work) can of course not be generated. This part of the implementation – still the major part today – is left to the programmers.

Creating the database can be automated to a significant extent. Studying the data model in figure 5-54, the reader may guess that most DDL code can be created automatically since all essential information for the data definitions is already included in the diagram. Tuning and refining the generated database code is then left to database programmers.

The third component of an information system, the graphical user interface, is usually also generated automatically. This is done with the help of the GUI toolbox in which the design was created. Code generation for graphical user interfaces will be illustrated in section 6.1.3.

6 Implementation and Testing

Implementation is a term with many meanings. This can lead to confusion when people from different disciplines use it. In organizational theory, implementation means putting some concept or plan into operation in a real organization. Computer scientists used to call the coding activities (i.e. writing programs) implementation. However, in a more general view, not only programs but any model or concept that is to be executed on a computer must be implemented. We will use the term with the following meaning:

"Implementation" has many meanings

Definition: implementation

Implementation is the realization of a design so that it can be executed on a computer. This includes the realization of: the system's classes; the user interface; and the database structures.

The classes and the user interface are implemented in a programming language whereas the database structures are implemented in a data definition language. Other concepts such as workflows may need to be implemented as well so that they can be executed automatically.

Testing means finding errors

Testing is an approach to finding errors in the system under consideration. It does *not* mean to ensure that the system is correct! Errors can occur on various levels: in the code, in the design, in the requirements or even in the problem statement. In any case, an error means that the system is not functioning as it should.

Definition: testing

Testing comprises all activities to accomplish a satisfactory level of confidence that the system under development fulfills it intended purpose. Objects of testing can be documents (such as specifications) or software (such as a module or a complete system). The goal of testing is to find errors and remove the causes of the errors.

Testing and implementation are obviously closely connected. If a concept is to be executed on a computer, as the definition of "implementation" states, then the implementation must run without errors, otherwise the concept has not been implemented (or at least not correctly implemented). In practice, implementation and testing go hand-in-hand. This aspect will be discussed later, in section 6.2. First we will consider the basic methods and tools for implementation.

6.1 Implementing the Design

In this section, some aspects of implementing a design are discussed. Implementation means primarily to use computerized tools: a programming language, a database management system (in particular its data definition language), and higher-level tools that may facilitate the implementation.

6.1.1 Programming

The core of implementation is to write programs. Previously, implementation was more or less a synonym for programming. The main task was to create Cobol, PL/1 or Assembler programs. Database programming was included, and user interfaces did not play any significant role.

Programming is not the only implementation task

Nowadays, writing code in a programming language is only one of several implementation tasks, yet still the most voluminous and time-consuming one. Some of the former tasks no longer apply because they are solved outside the normal programs (e.g. data definitions, transactions processing). On the other hand, additional tasks have to be solved that were not there before. Examples are network programming for distributed systems, synchronization of parallel tasks (thread programming) and socket programming for Internet based systems.

Structured programming (SP)

Programming methodology was an intensively discussed topic for many decades, initiated by Edsger Dijkstra's famous article "Goto statement considered harmful" [Dijkstra 1968a] that lead eventually to the emergence of *structured programming (SP)*. SP was about writing understandable and maintainable programs, focusing on simple control structures and on modularization. While the discussion about SP has come to an end, the principles of SP are commonly accepted today and SP has become general programming practice. Therefore we will not discuss programming methodology in this book.

Fundamental program structuring principles

More fundamental principles for proper program structures, in addition to those of structured programming, were proposed and also intensively discussed in the 1970s and 1980s. These principles include, in particular, information hiding and encapsulation, abstract data types (ADTs), and finally the concept of objects communicating via messages with each other. Eventually the principles and concepts were incorporated into the design of programming languages, so they are also considered common knowledge and practice today.

Programming languages

Programming paradigms

The primary tool for programming is a *programming language.* In the early times of computing, a programming language was a language in which the operations and data were specified, i.e. the operations to be performed, the sequence of the operations, and the data used as input and created as output. This was called procedural (or imperative) programming. Other programming styles (or programming paradigms) were introduced in the course of time, leading to a variety of programming languages. Figure 6-1 summarizes the most important paradigms and examples of corresponding languages.

Figure 6-1 **Programming paradigms and languages**

Programming paradigms	Programming languages
Procedural (imperative)	Fortran, Cobol, Pascal, …
Functional	Lisp, FP
Object-oriented	Java, C++, Ada95, Visual Basic .NET
Declarative	SQL
Logic	Prolog
Event-driven	Visual Basic .NET

Although hundreds of programming languages have been invented and many of them have survived, there seems to be a convergence towards a few widely used languages.

Java and Visual Basic are the dominating languages today

The dominating languages today are Java and Visual Basic. *Java* has become the most wide-spread language for large professional software systems across a broad range of application areas. *Visual Basic* is the preferred language in the Microsoft world. Since Microsoft is strong in PC applications, Visual Basic is mostly used for small and midsize software systems running in a Microsoft environment. Other languages with large developer communities include C++, C#, Prolog and Delphi. Proprietary languages with large user groups also exist (e.g. ABAP for SAP developers).

Java

Java was developed at Sun Microsystems by James Gosling and colleagues in 1995. The first public version was shipped as the JDK 1.0 ("Java development kit") in January 1996. Both the language and the development environments have undergone several renamings and numbering schemes. The language as such was called "Java 2" for some years, but as of 2007 the official name is "Java" again.

Java rapidly gained widespread acceptance because it implemented most object-oriented concepts that were considered desirable by the software engineering community. Java provides powerful mechanisms for inheritance, polymorphism, persistence and more. Since the language design is also object-oriented, Java can easily be extended. A large number of class libraries around the language core are available, including comprehensive GUI support (cf. section 6.1.3). Java is a cross-platform language, running under MS Windows, Apple OS X, Sun Solaris, Linux etc. Language versions are available for other devices than regular computers, e.g. for mobile phones and PDAs.

Visual Basic

Visual Basic has its roots in the old Basic language developed in 1964 for educational purposes at Dartmouth College in New Hampshire. *Basic* stood for "*B*eginners *a*ll-purpose *s*ymbolic *i*nstruction *c*ode", and that was exactly the goal: a language for beginners in which they could write instructions for the computer in a "symbolic" way (as opposed to writing machine-oriented code in Assembler). The original Basic language was very simple, providing only 14 statement types altogether.

Today's Visual Basic has little to do with that simple language of 1964 except for the name and some keywords that have survived. The name Visual Basic was introduced by Microsoft in 1987. The language came with a visual development environment, the first tool of its kind by Microsoft. After a few years, Visual Basic had become the fastest-growing programming language on the market.

Visual Basic gained its popularity mainly due to two reasons: 1) It was easy to use because the language was embedded in a powerful development environment that conveniently supported graphical user interface design, coding, and testing. 2) Visual Basic dialects were embedded in Microsoft Office programs such as Excel and Access, capable of extending the functionality of these programs.

Visual Basic .NET

While Visual Basic – up to the version Visual Basic 6 – was just an easy-to-use procedural event-oriented programming language, the introduction of Microsoft's .NET platform brought a fundamental redesign of the language. Visual Basic became a full-fledged object-oriented programming language with all the important object-oriented features such as classes, inheritance, polymorphism etc., similar to Java and C++. The name was changed to *Visual Basic .NET*.

The design of this language is completely different from Visual Basic 6 and earlier versions. It is based on the general concepts and mechanisms of the .NET framework (cf. section 3.5.2). That is why the other .NET languages such as C# are very similar to Visual Basic .NET. Although they look different (syntactically), the fundamental language concepts are the same.

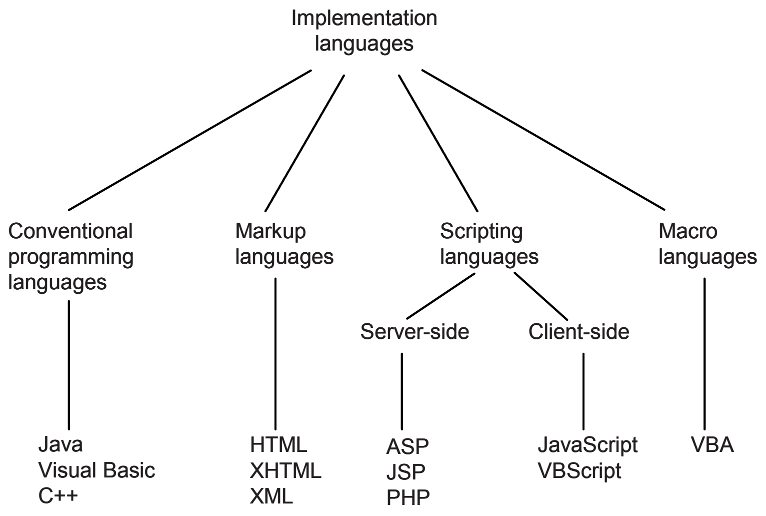
Visual Basic .NET is embedded in Visual Studio. This is a very powerful IDE providing the same functionality for all .NET programming languages.

Other languages

Other implementation languages

A conventional programming language – as above – is not the only type of language used in the implementation stage. Other languages are markup, scripting and macro languages as shown in figure 6-2.

Figure 6-2 **Languages for implementation of information systems**



Markup languages

A *markup language* is a language that uses markups to specify properties of text or text documents. Markup languages have become the primary means of creating front-ends for web based systems. The most important markup languages are HTML, XHTML (eXtensible HTML) and XML.

Scripting languages

Extensions of static web pages are developed with the help of *scripts*. Scripts are programs embedded in web pages (client-side scripts) or running on a web server (server-side scripts), written in a scripting language. A scripting language is usually embedded in a *scripting technology*, because it is not the naked language but the technology around that allows the language to access and manipulate web objects. For example, the scripting technology for server-side scripting in Microsoft systems is ASP (Active Server Pages), but the language used to write the scripts is Visual Basic. In environments working with JSP (JavaServer Pages), the scripts are small Java programs.

Some scripting technologies are available for server-side scripting only, others for client-side scripting. JavaScript is the most commonly used language for client-side scripting, while ASP, JSP and PHP are the major technologies for server-side scripting.

A *macro* is a piece of code similar to a script. Macros have been used in many contexts. Today, a macro usually stands for a sequence of steps that is initiated by a user action (such as pressing a button or a keyboard combination). Behind this button or keyboard combination is a program that executes the desired steps. Some macros are just a few lines of code, but others extend over many pages of program text.

Macro languages

A *macro language* is a language to write macros. Macro programming is very popular in and around Microsoft Office tools such as Excel, Access and Word. The programming language used here is VBA ("Visual Basic for Applications"). Voluminous information systems, especially for small enterprises, have been written using VBA inside Excel and Access. Likewise, many end-users develop their personal information systems in this way. The formatting of this book was done with the help of macros written in VBA that run inside MS Word.

Technology-supported programming

What is left to the programmers?

Not all program code needs to be written from scratch. Actually the opposite is true. Most code of a typical information system has been written by other people before or is generated automatically. What is left to the programmers in the implementation phase are four major tasks:

1. Generating code from diagrams, models and/or GUI designs, and extending or modifying the code. This is discussed below and in sections 6.1.2 and 6.1.3.
2. Writing application-specific code that cannot be generated.
3. Invoking prewritten code available through APIs (application programming interfaces) and/or including code from program libraries into the system under development. This will be discussed in section 6.2.3.
4. Testing code pieces as they are created, in parallel with the programming.

Although testing is usually considered as an independent set of activities (or an independent stage), the immediate testing of new code pieces is so closely intermingled with the writing of these code pieces that it cannot reasonably be separated. Therefore we include this type of testing (actually: module testing, cf. section 6.3.4) as a programmer's responsibility.

Generating code from class diagrams

Code generation from class diagrams is a standard feature of CASE tools. The result consists of such code that could be automatically derived from the information available in the design class diagrams.

Generated Java code for "Invoice" and "InvoiceItem"

Generated Java code for the two classes "Invoice" and "InvoiceItem" of figure 5-53 is shown in figures 6-3 to 6-6. The CASE tool that produced the code is the same one we used before (Enterprise Architect). Attributes contained in the DCD were adopted as private variables in the class definitions. Some new variables were also created by the tool (prefixed by "m_"). These variables are needed to reference other objects. For example, in the "Invoice" class there are new "m_Customer" and "m_InvoiceItem" variables through which the respective "Customer" and "InvoiceItem" objects can be accessed.

Java method stubs were also generated. Since the DCD does not contain much information regarding methods, only the methods' signatures (method name, parameters, return types), empty comments for the parameters and return statements in the methods' bodies could be derived automatically. The programmer is expected to extend, modify and/or delete the generated code, writing program statements and comments.

Constructors, getter and setter methods

The tool generated parameterless constructors, in addition to the parametrized constructors defined in the DCD. So-called *getter* and *setter methods* were also created[§]. Some CASE tools generate these methods by default, others do so only if these methods are explicitly specified. (Our CASE tool required us to explicitly specify getters and setters in the design class diagram. The stubs that were generated from this extension are shown in figure 6-4.)

Code generation for the two classes "Invoice" and "InvoiceItem" (cf. figures 6-3 and 6-5) was satisfactory insofar as the tool made no real mistakes. One missing point is, however, that no reference from the "InvoiceItem" class to the "Invoice" class was created. A reference was only generated in the other direction (variable "m_InvoiceItem").

§ Getter and setter methods are methods used to access the values of private variables. According to the information-hiding and encapsulation priniciples, other objects should not be allowed to access internal details of an object's implementation. A common solution is instead to define a method returning the object's value ("get") and a method to set the object's value ("set").

Figure 6-3 **Generated Java code for "Invoice" class**

```
/**
 * @version 1.0
 * @created 27-Dec-2007 10:25:14
 */
public class Invoice {

    private IDType invoiceID;
    private Date date;
    private Employee salesRep;
    public Collection m_InvoiceItem;
    public Customer m_Customer;

    public Invoice(){

    }

    /**
     *
     * @param customer
     */
    public Invoice(Customer customer){

      }

    public void finalize() throws Throwable {

      }

    /**
     *
     * @param invoiceItem
     */
    public void addItemLine(InvoiceItem invoiceItem){

    }

    /**
     *
     * @param invoiceItem
     */
    public void deleteItemLine(InvoiceItem invoiceItem){

    }

    public double getTotal(){
        return 0;
    }

    public String printInvoice(){
        return "";
    }

}
```

Figure 6-4 **Getter and setter methods for "Invoice" class**

```
    public IDType getInvoiceID(){
        return null;
    }

    /**
     *
     * @param newId
     */
    public void setInvoiceID(IDType iDType){

    }

    public Date getDate(){
        return null;
    }

    /**
     *
     * @param date
     */
    public void setDate(Date date){

    }

    public Employee getSalesRep(){
        return null;
    }

    /**
     *
     * @param employee
     */
    public void setSalesRep(Employee employee){

    }
```

Therefore the programmer will need to manually add to the code of "InvoiceItem" a declaration such as the following:

```
public Invoice m_Invoice;
```

Generated code must be reviewed carefully

This is only a minor change, but it shows that generated code should be reviewed carefully. ("public" is a Java modifier created for association variables by the CASE tool. With respect to information hiding and encapsulation, "public" is actually not appropriate and should be changed to "private", as in the next example.)

A more severe example where code revision is indispensable is the code generated for the "Advertisement", "Edition" and "Publication" classes (cf. figure 6-6). The latter class is actually an association class connecting the other two classes.

Figure 6-5 **Generated Java code for “InvoiceItem” class**

```
/**
 * @version 1.0
 * @created 27-Dec-2007 10:55:29
 */
public class InvoiceItem {

    private IDType itemID;
    private double price;
    private Advertisement advertisement;
    private Edition edition;
    public Publication m_Publication;
    public Collection m_Advertisement;
    public Collection m_Edition;

    public InvoiceItem(){

    }

    public void finalize() throws Throwable {

    }

    /**
     *
     * @param advertisement
     * @param edition
     * @param double
     */
    public InvoiceItem(Advertisement advertisement,
                       Edition edition, price double){

    }

    /**
     *
     * @param publication
     */
    public InvoiceItem(Publication publication){

    }

    public double getPrice(){
        return 0;
    }

}
```

What the CASE tool generated, however, is an unsatisfactory mapping. In the "Advertisement" class, a variable to reference "Edition" objects was created ("m_Edition"), but this is only a one-directional mapping. Neither did the tool generate variables representing the connections from "Advertisement" objects and from "Edition" objects to "Publication" objects.

Figure 6-6 **Generated code for "Advertisement", "Edition" and "Publication"**

```
/**
 * @version 1.0
 * @created 27-Dec-2007 11:11:54
 */
public class Advertisement {

    private IDType adID;
    private String adName;
    private String description;
    private String fileName;
    public Collection m_Edition;
    ...
}

public class Edition {

    private IDType edId;
    private String volume;
    private String issue;
    private Date adsDeadline;
    private Date date;
    ...
}

public class Publication {

    private Advertisement advertisement;
    private Edition edition;
    private int state;
    private double price;
    ...
}
```

The programmer has to write a proper mapping

The proper mapping would have to be written by the programmer. The new code is shown in figure 6-7 (additional statements in italics). Three major changes are: 1) Association variables "m_Publication" were introduced in "Edition" and "Advertisement" as private variables. 2) The generated public association variable "m_Edition" in "Advertisement" was discarded. 3) Two methods "addPublication" to fill the collections "m_Publication" in "Edition" and "Advertisement" were added.

More methods to handle the generated and the additional association variables have to be written by the programmer. We refrain from discussing more coding details since the topic of this book is not Java programming.

Most tools produce incomplete mappings

Although it would be nice if the CASE tool generated code such as the one in figure 6-7 (plus additional extensions mentioned) automatically, most state-of-the-art tools produce incomplete mappings similar to the code shown in figure 6-6.

Figure 6-7 **Modifications of generated code**

```
/**
 * @version 1.0
 * @created 27-Dec-2007 11:25:51
 */
public class Advertisement {

    private IDType adID;
    private String adName;
    private String description;
    private String fileName;
    private Collection m_Publication;
    ...

    public void addPublication(Publication publication) {
    ...
    }
}

public class Edition {

    private IDType edId;
    private String volume;
    private String issue;
    private Date adsDeadline;
    private Date date;
    private Collection m_Publication;
    ...

    public void addPublication(Publication publication) {
    ...
    }
}

public class Publication {

    private Advertisement advertisement;
    private Edition edition;
    private int state;
    private double price;
    ...
}
```

6.1.2 Implementing the Database

Provided that the information system under development will employ a relational database – as the majority of today's IS do – implementing the database means primarily to create database schemata in a DDL (data definition language). The dominating language for relational databases is SQL (structured query language). Therefore the data definitions have to be formulated in SQL, in particular in "create table" statements.

We will illustrate the data definitions for the advertisement-ordering system with SQL statements, assuming that the reader has some basic knowledge of SQL. Readers who do not know SQL are advised to consult an introductory text on SQL [e.g. van der Lans 2006], or skip this section.

CASE tools generate complete SQL data definitions

Since the UML data model we created with our CASE tool before contained plenty of implementation oriented detail, generating complete and correct SQL data definition statements is straightforward. The result is shown in figure 6-8. (Some line breaks were removed from the generated code to make it fit into the print area of this book.) The code was generated for an Oracle database, because we selected this DBMS before (cf. section 5.3.2).

The first portion of the generated SQL code contains "drop" statements for all tables, meaning that tables with the same names as existing tables are discarded. The "create table" portion mirrors the attributes of the respective tables in the data model of figure 5-54.

Primary keys are specified as constraints

The generator did not place primary-key clauses directly with the respective attributes, such as

```
CREATE TABLE Advertisement
  (adID VARCHAR(32) PRIMARY KEY NOT NULL ...;
```

but specified all primary keys as constraints on the respective tables with the help of "alter" statements. Figure 6-8 shows three of the nine generated "alter" statements – one for each "create" statement. (The rest have been omitted from the figure.)

Foreign keys were also not declared directly via foreign-key clauses, but specified as constraints; for example:

Figure 6-8 **Generated DDL code for advertisement-ordering system**

```
DROP TABLE Advertisement CASCADE CONSTRAINTS;
DROP TABLE Agency CASCADE CONSTRAINTS;
DROP TABLE CommissionScheme CASCADE CONSTRAINTS;
DROP TABLE Customer CASCADE CONSTRAINTS;
DROP TABLE Edition CASCADE CONSTRAINTS;
DROP TABLE Invoice CASCADE CONSTRAINTS;
DROP TABLE InvoiceItem CASCADE CONSTRAINTS;
DROP TABLE Publication CASCADE CONSTRAINTS;
DROP TABLE RegularCustomer CASCADE CONSTRAINTS;

CREATE TABLE Advertisement (
    adID VARCHAR(32) NOT NULL, adName VARCHAR(120),
    description CLOB, fileName VARCHAR(255));

CREATE TABLE Agency (
    custID VARCHAR(32) NOT NULL, category VARCHAR(50),
    contactPerson VARCHAR(120), comID VARCHAR(32));

CREATE TABLE CommissionScheme (
    comID VARCHAR(32) NOT NULL, description CLOB);

CREATE TABLE Customer (
    custID VARCHAR(32) NOT NULL, name VARCHAR(120),
    address VARCHAR(255), phoneNumber VARCHAR(28));

CREATE TABLE Edition (
    edID VARCHAR(32) NOT NULL, volume VARCHAR(50),
    issue VARCHAR(50), date DATE);

CREATE TABLE Invoice (
    invoiceID VARCHAR(32) NOT NULL, custID VARCHAR(32),
    date DATE, salesRep VARCHAR(32));

CREATE TABLE InvoiceItem (
    itemID VARCHAR(32) NOT NULL, adID LONG, edID LONG,
    invoiceID VARCHAR(32), itemPrice NUMBER(8,2),
    quantity NUMBER(8,2));

CREATE TABLE Publication (
    adID VARCHAR(32) NOT NULL, edID VARCHAR(32) NOT NULL,
    custID VARCHAR(32) NOT NULL, invoiceItemID VARCHAR(32) NOT NULL,
    state VARCHAR(50));

CREATE TABLE RegularCustomer (
    custID VARCHAR(32) NOT NULL,
    discountScheme VARCHAR(50));

ALTER TABLE Invoice ADD CONSTRAINT PK_Invoice
    PRIMARY KEY (invoiceID);

ALTER TABLE InvoiceItem ADD CONSTRAINT PK_InvoiceItem
    PRIMARY KEY (itemID);

ALTER TABLE Publication ADD CONSTRAINT PK_Publication
    PRIMARY KEY (adID, edID);

  ...

ALTER TABLE Invoice ADD CONSTRAINT FK_Invoice_Customer
    FOREIGN KEY (custID) REFERENCES Customer (custID);

ALTER TABLE InvoiceItem ADD CONSTRAINT FK_InvoiceItem_Invoice
    FOREIGN KEY (invoiceID) REFERENCES Invoice (invoiceID);
```

```
ALTER TABLE InvoiceItem ADD CONSTRAINT FK_InvoiceItem_Invoice
   FOREIGN KEY (invoiceID) REFERENCES Invoice (invoiceID);
```

Seven "alter" statements like the above were generated – one for each foreign-key relationship. Two of them are included in the figure.

The database is ready to use. Java classes of the advertisement-ordering system could access the database via a DML (data manipulation language) directly or through middleware such as JDBC (Java database connectivity). Database generation from data models is the part of code generation that has always worked best, even with the CASE tools of the early 1990s.

6.1.3 Implementing the User Interface

Regarding the user interface, design and implementation activities are closely connected. The reason for this is that user interfaces today are usually designed in a prototyping approach. When CASE tools, IDEs or GUI toolboxes are employed to develop a GUI prototype, the result is available as executable code – notwithstanding the fact that the user interface was probably created by simple drag-and-drop.

What is missing in such a prototype is the functionality behind the GUI elements, e.g. what happens when an item from a listbox is selected or a button is pressed? These *events* caused by the user must be handled in the program, requiring that code is written – so-called event handlers.

GUI oriented design and implementation

When the user-interface plays a dominant role in the system under development, GUI design and implementation can even be the initial activities – before the rest of the system is implemented. An event-driven, object-oriented process submodel for design and implementation would place GUI creation before implementation of the non-GUI classes and the database:

Event-driven subprocess model

1. Design the user interface (i.e. generate, position and size GUI objects; think about possible events).
2. Define relevant properties and methods (event handlers) for the GUI objects.

3. Design the non-GUI classes of the problem domain and the database.
4. Implement event handlers to be executed when a GUI event occurs.
5. Implement the non-GUI classes, especially the methods to be invoked by GUI event handlers, and the database.

Event handlers

In the case that the user interface is HTML based, an *event handler* would be a script. In a Windows form, it is a Visual Basic .NET method (a "sub" procedure). In a Java form, an event handler is a Java method.

An example

As an example, consider a registration form for online courses provided by a seminar company. The form contains a number of form elements (controls) where clients can enter their data and course selections, including the following:

– Text fields for the client's first name, last name and e-mail address (with HTML names "txtFirstName", "txtLastName", "txtEmail")
– Check boxes for the seminars to select from (with HTML names "chkBi101", chkMoIS" and "chkWiF")
– Submit, reset and print buttons

Several GUI events can happen in this form, for example: text is entered into a text field; a check box is marked; one of the three buttons is pressed, etc. Suppose the event "submit button is pressed" occurs. An event handler dealing with this event will be invoked. It needs to check whether the data entered are plausible and then submit the form to the web server for processing.

A JavaScript event handler

Figure 6-9 shows a simplified excerpt of an HTML document containing the web form code. When the user fills out the form and presses the submit button on the GUI, the "onSubmit" event occurs. Within the <form> tag, an event handler associated with the "onSubmit" event is invoked. The event handler in this case is a JavaScript function named "checkInput". If it returns true, the form is sent to the web server where it will be processed by a PHP script called "registration.php".

Some of the code has been omitted in the figure, in particular the definitions of the form elements (controls). Names of the form elements are used in the event-handler code. In figure 6-9, these are the text-field names ("txtFirstName") and check-box names (e.g. "chkBi101").

The above example shows that event handling in JavaScript is quite simple. In Visual Basic it is both simple and powerful, whereas in Java a lot of work before and around the actual handling has to be done. On the other hand, Java provides the most powerful event-handling facilities.

The question of how easy or difficult it is to develop a graphical user interface depends not only on the language but also on the desired flexibility. On a scale from static to dynamic, three major categories of user interfaces can be distinguished:

– completely static
– semi-static
– completely dynamic

Figure 6-9 **JavaScript event handler in a web form**

```
<html>
<head>
  <title>Registration Form</title>
  <script type="text/javascript">    <!--

   function checkInput() {
     with(document.regForm) {
       if(txtFirstname.value.length == 0)
         alert("Your first name is missing");
       else if(txtLastname.value.length == 0)
         alert("Your last name is missing");
       else if((txtEmail.value.indexOf("@") < 0) ||
               (txtEmail.value.indexOf(".",
                txtEmail.value.indexOf("@")) < 0))
         alert("Your e-mail address is wrong!");
       else if(!(chkBi101.checked || chkMoIS.checked ||
                chkWiF.checked))
         alert("You did not register for any course!");
       else {
         var result;
         result = confirm("Send your data?");
         return result;
       }
       return false;
    }
   }                                 //-->
  </script>
</head>

<body>
  <form name="regForm" method="get"
        action="registration.php" enctype="text/plain"
        onSubmit="return checkInput()">

    ...

  </form>
</body>
</html>
```

Completely static GUIs

A *completely static GUI* is one where all form elements are placed on the user interface and all properties are defined when the GUI is designed and implemented. Developing such a user interface is easy when it is done "manually", i.e. with the help of a GUI tool. In this case the designer can employ the drag-and-drop features provided by the tool.

The actual program code that will create the user interface (i.e. the screen layout and the controls) when the program is executed is generated behind the screen, along with the designer's drag-and-drop actions. The GUI is (more or less) fixed in the sense that it is displayed exactly as it was manually designed. Figure 6-9 described a completely static GUI§.

Semi-static GUIs

A *semi-static GUI* is created by the designer in the same way as above. However, when the program executing the GUI code is running, some properties and values of GUI elements (e.g. form elements such as text fields, list boxes etc.) are assigned dynamically, or the GUI may even be reconfigured dynamically (e.g. adding new elements, omitting pre-designed ones). If there are several options of what to display in a form and how to display it, then a designer could create several forms and allow the program to choose the appropriate alternative at runtime.

The developer needs to know the details of the available GUI controls in order to be able to fill them with content and tailor their appearance. In an object-oriented programming language, this usually means that the developer must know the object model underlying the user-interface software (e.g. the HTML document object model used by the web browser) and the methods and properties of the generated objects.

Completely dynamic GUIs

A *completely dynamic GUI* is created by a program during runtime, usually from code that was written by a human developer and not generated by a tool. Dynamic GUIs are required when the elements and/or contents of the user interface depend on the user, the state of the program or the data. An example where dynamic GUI creation is needed is personalized web pages.

Writing a program that entirely defines a user interface through text (program statements) by hand is a cumbersome and difficult programming task. The developer needs to know not only the available properties and methods of generated objects, as in the semi-static case, but also how to create the objects. In an object-oriented programming language,

§ With regard to behavior resulting from user actions, the GUI underlying figure 6-9 is in fact dynamic. Plausibility checks are done, and submission of the form will result in some server action. However, from the perspective discussed here – creating the form itself – the GUI is a completely static one.

this usually means that the developer must know the available GUI packages, in particular the classes from which concrete user-interface objects can be instantiated or from which subclasses can be derived (e.g. the Java Swing classes and APIs).

Manually writing GUI code is awkward

The reader can imagine how much work it takes to write GUI code directly in a programming language by looking at the example in figure 6-12. This code was generated with the help of a visual GUI editor and a built-in code generator. Suppose the same code had to be developed without such tools. Then the developer would need to create all the objects, attributes and methods through manually writing Java statements. That this can be avoided illustrates how powerful today's development tools for graphical user interfaces actually are.

Generating a graphical user interface

In the following example, creating a GUI with the help of a GUI editor and a code generator is demonstrated. The subject is a development of a user form for the advertisement-ordering system through which customers can place orders for publication of advertisements.

A user form for ordering advertisements

Figure 6-10 shows one of the forms provided for customer interaction. It is assumed that a login form, or a form for registration of new users, redirected the customer to the current form. What a customer should be able to do now are three things:

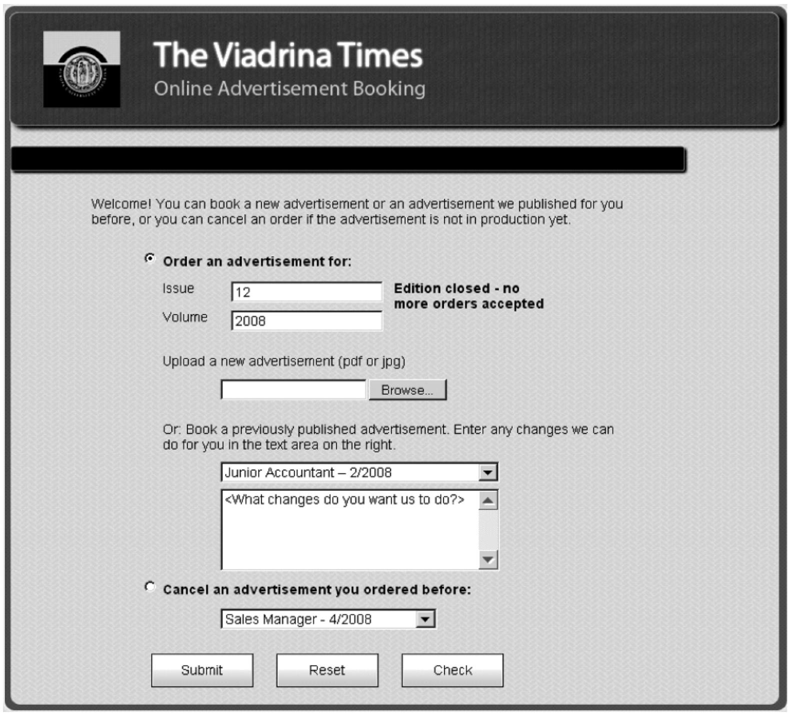
1. Place an order to publish a new advertisement. In this case, the customer should be assisted in finding an advertisement file on his or her computer and then uploading the file.
2. Place a repeat order for an existing advertisement. The customer should be allowed to select from a list of previously published advertisements. A drop-down list as shown in figure 6-10 ("Junior Accountant – 2/2008") provides this information. In the text area underneath the drop-down list, the customer can write instructions regarding how the previous advertisement should be amended.
3. Withdraw an already booked order. In this case, open orders which can still be cancelled (i.e. not in production yet) are provided in another drop-down list ("Sales Manager – 4/2008").

In the screenshot shown in figure 6-10, the customer was going to book an advertisement for newspaper edition 12/2008 ("Issue = 12", "Volume = 2008"). The advertisement he wanted is one that was published before in this newspaper ("Junior Accountant – 2/2008"). Unfortunately the deadline for booking advertisements for edition 12/2008 was already over. The ordering system noted this when the customer pressed the "Check" button, and produced an error message ("Edition closed – no more orders accepted.")

NetBeans and NetBeans Visual Web Pack

The GUI in figure 6-10 was created with an integrated development environment for Java, *NetBeans IDE*, and an add-on for web development, NetBeans Visual Web Pack. (Both are available for download from http://www.netbeans.org.) We chose a web GUI and not a Java Swing GUI because more and more business information systems provide web front-ends for their users instead of proprietary front-ends.

Figure 6-10 **Web GUI for advertisement-ordering system**



The Viadrina Times
Online Advertisement Booking

Welcome! You can book a new advertisement or an advertisement we published for you before, or you can cancel an order if the advertisement is not in production yet.

◉ **Order an advertisement for:**

Issue: 12 **Edition closed - no more orders accepted**

Volume: 2008

Upload a new advertisement (pdf or jpg)

Browse...

Or: Book a previously published advertisement. Enter any changes we can do for you in the text area on the right.

Junior Accountant – 2/2008

<What changes do you want us to do?>

○ **Cancel an advertisement you ordered before:**

Sales Manager - 4/2008

Submit Reset Check

The GUI was largely created by drag-and-drop, with some properties edited with the tool's property editor. In particular, the names of the GUI objects and most of the text shown in the figure were defined in this way. The main purpose of the example is to show different GUI elements and the code generated behind the curtain for these elements. Since the example and the IDE are Java oriented, the generated code is in the first place JSP (JavaServer Pages) and Java code.

Figure 6-11 shows JSP code that the NetBeans add-on generated from the GUI design. The purpose of this figure is not to explain vendor-specific XML code – NetBeans was originally a product of Sun Microsystems before it became open source – but to illustrate how much coding work a GUI tool does for the programmer. If this code had not been generated from the visual design, the programmer would need to type similar code manually instead.

Generated JSP code

The code in the figure contains only the part of the JSP page where the order form is specified. The page actually begins with XML information about the namespace[§] used, the page structure etc., and it ends with the respective closing tags. Many tags have attributes with much more content than what can be displayed in one line of this book. That is why the text is cut off at the right-hand side.

Figure 6-12 shows excerpts of the *Java code*. The figure was edited somewhat to make it fit on a book page, and most of the 15 pages of generated code was deleted. Lines containing only a colon indicate omissions. Again, we are not going to explain details of Java programming but only point out some essential features.

The Java code of the advertisement-ordering system is contained in a package named "advertisement_ordering", generated by NetBeans. All the GUI components are imported from libraries. For example, the

```
import com.sun.rave.web.ui.component.TextField;
```

statement imports the class "TextField" from the "com.sun.rave.web.ui.component" library package. An object with the name "orderForm" with a setter and a getter was created from the specification of this object in the GUI design. (The name "orderForm" was set as the value of the "id" property of the form object; cf. figure 6-14 below.)

From the various GUI components of the form, generated code for three of them is visible in the boxes on the left-hand side and on the upper right-hand side of figure 6-12: the text field named "txtIssue" in which the user can enter the issue of the newspaper, the drop-down list named "drpPreviousAds1" containing previous advertisement orders, and the button named "checkBtn" allowing the user to check whether the intended booking is possible.

§ The namespace has the name "ui" which is used to prefix NetBeans custom tags. For example, the <ui:button> tag is NetBean's implementation of a button in JSP.

Figure 6-11 **Generated JSP code for advertisement-ordering GUI**

```
  :
<ui:form binding="#{MainPage.orderForm}" id="orderForm">
  <h1>
    <img alt="The Viadrina Times - Online Advertisement Booking" height="80" s
  </h1>
  <div id="Sep1"> </div>
  <div>
    <div id="Menu"> </div>
    <div id="Main">
      <p>
        <ui:staticText binding="#{MainPage.staticText1}" id="staticText1" text
      </p>
      <ul class="radios">
        <li>
          <ui:radioButton binding="#{MainPage.radOrder}" id="radOrder" label="
        </li>
        <div class="list">
          <p>
            <ui:label binding="#{MainPage.lblIssue}" for="txtIssue" id="lblIss
            <ui:textField binding="#{MainPage.txtIssue}" id="txtIssue" text=""
          </p>
          <p>
            <ui:label binding="#{MainPage.lblVolume}" for="txtVolume" id="lblV
            <ui:textField binding="#{MainPage.txtVolume}" id="txtVolume" text=
          </p>
          <br/>
          <p>
            <ui:label binding="#{MainPage.lblUpload}" for="upldFileUploadField
          </p>
          <p>
            <ui:upload binding="#{MainPage.upldFileUploadField1}" columns="19"
          </p>
          <br/>
          <p>
            <ui:label binding="#{MainPage.lblRepeat}" for="drpPreviousAds1" id
          </p>
          <p>
            <ui:dropDown binding="#{MainPage.drpPreviousAds1}" id="drpPrevious
          </p>
          <p>
            <ui:textArea binding="#{MainPage.tarChanges}" columns="40" id="tar
          </p>
        </div>
        <li>
          <ui:radioButton binding="#{MainPage.radCancel}" id="radCancel" label
        </li>
        <div class="list">
          <ui:dropDown binding="#{MainPage.drpPreviousAds2}" id="drpPreviousAd
          <br/>
          <br/>
        </div>
        <li>
          <ui:button binding="#{MainPage.submitBtn}" id="submitBtn" text="Subm
          <ui:button action="#{MainPage.resetBtn_action}" binding="#{MainPage.
          <ui:button action="#{MainPage.checkBtn_action}" binding="#{MainPage.
        </li>
      </ul>
    </div>
  </div>
</ui:form>
  :
```

Figure 6-12 **Java code for advertisement-ordering GUI (1)**

```
package advertisement_ordering;

import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.DropDown;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.ImageComponent;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import com.sun.rave.web.ui.component.RadioButton;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextArea;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Upload;
import com.sun.rave.web.ui.model.Option;
import com.sun.rave.web.ui.model.SingleSelectOptionsList;
import de.uniffo.eabook.example.Edition;
import java.text.DateFormat;
import javax.faces.FacesException;
import javax.faces.event.ValueChangeEvent;

public class MainPage extends AbstractPageBean {
    // <editor-fold desc="Managed Component Definition">

      :

    private Form orderForm = new Form();

    public Form getOrderForm() {
        return orderForm;
    }

    public void setOrderForm(Form f) {
        this.orderForm = f;
    }

      :

    private TextField txtIssue = new TextField();

    public TextField getTxtIssue() {
        return txtIssue;
    }

    public void setTxtIssue(TextField tf) {
        this.txtIssue = tf;
    }

      :
```

Figure 6-12 **Java code for advertisement-ordering GUI (2)**

```
    private DropDown drpPreviousAds1 = new DropDown();

    public DropDown getDrpPreviousAds1() {
        return drpPreviousAds1;
    }

    public void setDrpPreviousAds1(DropDown dd) {
        this.drpPreviousAds1 = dd;
    }

      :

    private Button checkBtn = new Button();

    public Button getCheckBtn() {
        return checkBtn;
    }

    public void setCheckBtn(Button b) {
        this.checkBtn = b;
    }

      :
```

```
      :

    public String checkBtn_action() {
        String volume = (txtVolume.getText() != null)
            ? txtVolume.getText().toString() : "";
        String issue = (txtIssue.getText()) != null
            ? txtIssue.getText().toString() : "";
        Edition ed = Edition.getEdition(volume, issue);

        if (ed == null) {
            errMsgText.setText("No such edition - " +
                "please check issue and volume");
        }
        else if (!ed.isBeforeDeadline()) {
            errMsgText.setText("Edition closed - " +
                "no more orders accepted");
        }
        else {
            String sDate = DateFormat.getInstance().format
                (ed.getDeadline());
            errMsgText.setText(
                "Ordering open - deadline is: " + sDate);
        }

        return null;
    }
}
```

Figure 6-12 also shows some code the programmer added to the generated code. The second box on the right-hand side contains event-handler code for the "checkBtn" button. This code was executed when we created the screenshot of figure 6-10. In particular, the "else if" branch of the "if" statement produced the error message "Edition closed – no more orders accepted."

6.2 Lower CASE

Following the distinction between upper and lower CASE introduced in section 5.3, tools supporting the programming and testing stages fall into the category of *lower CASE*. Historically, many lower CASE tools started around a programming language, providing editing and debugging facilities in addition to a compiler and a runtime environment for the language. An example of a lower CASE tool with which many students previously started their programming careers is Turbo Pascal. Today a large number of toolsets – both language-specific and language-independent – are available for programming and testing activities.

6.2.1 Integrated Development Environments (IDEs)

Lower CASE tools are usually called IDEs (integrated development environments) today. An IDE can be characterized as follows:

Definition: IDE

An *IDE (integrated development environment)* is a set of interlocking tools for the development of programs. Core tools are tools supporting programming and testing. Additional tools may be included, supporting design and management tasks.

A summary of typical core tools and additional tools extending the core functionality is given in figure 6-13. In the core of an IDE are the following tools:

IDE toolset

– A *text editor* "understanding" the programming language used. Such an editor is capable of checking the syntax, and to a certain degree the semantics, of program statements entered by the developer.
– A *GUI toolbox* supporting the creation of graphical user interfaces. The toolbox provides a collection of controls that can be arranged by drag-and-drop features on a design pane. GUI code can be generated automatically from the design.
– A *compiler* or an *interpreter* for each programming language supported by the IDE.
– A *build-automation tool* (e.g. linker, binder, linkage editor) assembling all needed machine-language modules (compiled application program modules and library modules) into one executable program.
– A *debugger* tracing program errors and pointing out to the developer what is wrong at what position of the source code.

Figure 6-13 **Typical tools of an IDE**

Core IDE Tools	Additional Tools
Text editor	Design tools
GUI toolbox	Class browser
Compiler/interpreter	Forms designer
Linker, binder	Version-control system
Debugger	Project-management tools

IDE user interface

The user interface of an IDE (NetBeans) is shown in figure 6-14. This IDE was used to create the GUI of the advertisement-ordering system in figure 6-10. Five panes are currently open and visible in the screenshot:

- *Project window* – displaying the structure of the "Advertisement Ordering" project on the left, with web pages, JavaBeans, libraries etc.
- *GUI builder window* – the working area in the middle where the GUI design takes place. The developer is currently working on the design of the "MainPage.jsp" page.
- *Palette window* – the widgets provided by the GUI toolbox on the right-hand side, e.g. labels, text fields, buttons etc. They can be dragged and dropped onto the design pane and positioned wherever they are needed.

- *Outline window* – providing a hierarchical view of the objects; in the screenshot: displaying the document structure.
- *Properties window* – displaying properties of the objects. When the screenshot was taken, the "orderForm" object (i.e. the entire form for ordering advertisements) hightlighted in the outline window was the active object. Therefore the properties of this object are shown in the window.

Setting properties

The properties window allows the developer to simply enter an object's properties instead of assigning them in complicated JSP or Java code. For example, the form object in figure 6-14 received its name "order-Form" in such a way that the text string "orderForm" was typed in the "id" property field of the properties window at the right-hand side (thus overwriting the generated default name "Form1.")

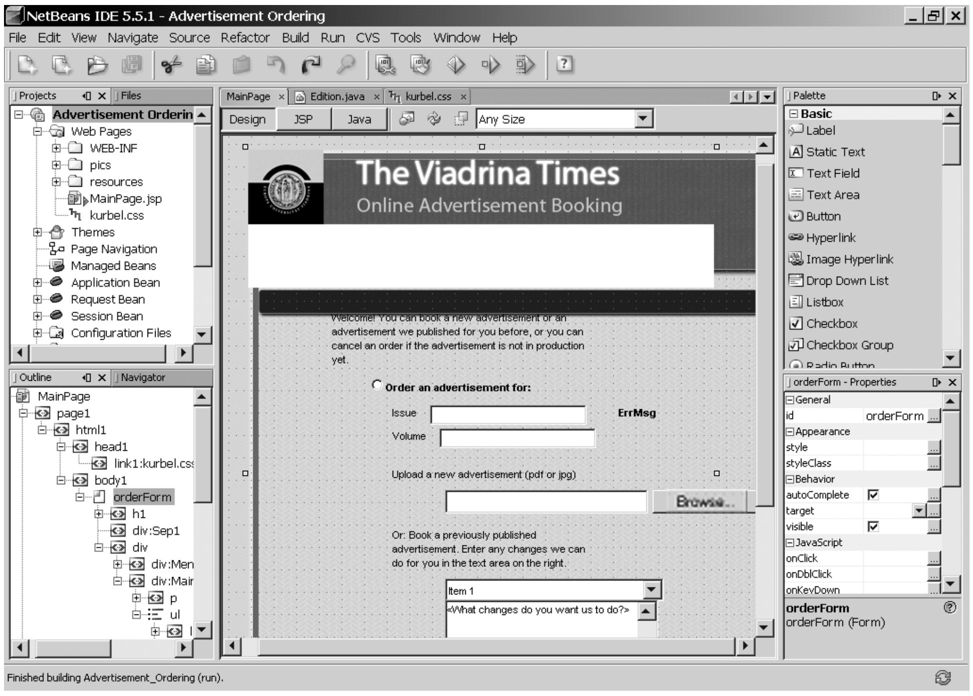
The NetBeans IDE contains more tools than the ones visible in figure 6-14. Some will be used further below, in particular the text editor, compiler and debugger (cf. section 6.3.5).

Additional IDE tools

While a text editor, GUI toolbox, compiler or interpreter, linker/binder and debugger represent a minimal set of tools found in all IDEs, comprehensive systems on the market offer a wide range of additional tools as listed in figure 6-13:

- *Program design tools* – providing simple or moderate support for the creation of UML diagrams, or for importing UML diagrams created by an upper CASE tool. This means that the IDE understands UML diagrams and is able to generate code from the diagrams.
- A *class browser* – visualizing the structure of an object-oriented system (e.g. in the form of class hierarchies) and allowing inspection of the classes' members.
- A *forms designer* as a special type of GUI tool – supporting the creation of forms-based user interfaces common in business information systems.
- A *version-control system (*VCS*)* – keeping track of various versions of modules, subsystems and the whole system when several developers are working in parallel (on different or on the same parts of the system). For example, such a tool time-stamps and records all changes submitted.
- *Project-management* features can be embedded in or connected with an IDE. For example, the Eclipse IDE provides a large number of plug-ins for project management.

Figure 6-14 **An IDE user interface (NetBeans)**



Welcome! You can book a new advertisement or an advertisement we published for you before, or you can cancel an order if the advertisement is not in production yet.

Upload a new advertisement (pdf or jpg)

Or: Book a previously published advertisement. Enter any changes we can do for you in the text area on the right.

<What changes do you want us to do?>

Finished building Advertisement_Ordering (run).

Version-control and project-management components embedded in an IDE usually do not reach the power of dedicated systems for these tasks. However, an IDE together with dedicated version-control and project-management systems can be a very powerful tool combination for software-development projects, provided that the tools are integrated. (Dedicated version-control and project-management systems will be discussed in chapters 8 and 9.)

Lower CASE tools assist programmers in many ways. The most obviously felt support is for editing and debugging. Along with the typing of program code, most tools immediately check the syntax of each keyword, identifier or text line entered, make auto-completion suggestions, and provide useful additional tips. Debugging with an IDE is very convenient, raising the developer's productivity significantly. This will be demonstrated in section 6.3.5.

Well-known IDEs

Java IDEs

An example of a Java-oriented IDE is *NetBeans*, the tool we used for GUI design and implementation in the previous section. Other Java IDEs, as mentioned in section 3.5.1, are Sun Java Studio, Microsoft J++, JBuilder, Eclipse and WebSphere Studio Application Developer.

Visual Studio .NET

In the Microsoft .NET world, the dominating IDE is *Visual Studio .NET* (http://www.microsoft.com/vstudio). This is a very powerful environment for all major .NET programming languages, including Visual Basic, Visual C++, Visual C#, and Visual J#.

Other multi-language IDEs

Well-known *multi-language IDEs* are Eclipse (http://www.eclipse.org), supporting Java, C, C++, Fortran, Cobol and scripting languages, and Borland Developer Studio (http://www.borland.com), covering C, C++, C# and Delphi/Object Pascal.

6.2.2 Connecting Upper and Lower CASE

Unfortunately the dream of I-CASE (*integrated* CASE – tool integration across all stages and activities of the software life cycle) did not come true, as mentioned in section 5.3. Today we have powerful upper CASE tools and powerful lower CASE tools.

What happens with the analysis and design results?

The reader may wonder how upper and lower CASE tools come together. An essential question for programming is: What happens with the results created with an analysis and design tool, in particular with the class diagrams? Are they put in the drawer, or can they be used – in electronic form – as input for the next activities?

Upper and lower CASE tools are overlapping

We already saw some overlap of upper and lower CASE in the previous examples: The lower CASE tool (NetBeans) was employed to *design* a GUI by drag-and-drop, from which Java code was generated. On the other hand, the upper CASE tool (Enterprise Architect) produced Java stubs and SQL statements, i.e. *code* for implementation. However, this is the end of the upper CASE tool's functionality.

Suppose the project under consideration employs Enterprise Architect (EA) for analysis and design, and NetBeans for implementation and testing. The Java code generated from the design class diagram is inside EA, but now we have to continue with NetBeans.

Some IDEs provide import features allowing the developer to import models (diagrams) created with a different tool. For example, since Rational Rose used to be a common toolset for analysis and design, some IDEs such as Visual Studio and JBuilder allowed Rose UML models to be imported. Plug-ins are often provided to do this work.

Upper and lower CASE tools are not integrated

Apart from such point-to-point import features, the situation regarding upper and lower CASE integration is rather disillusioning. What many programmers do in order to avoid starting from scratch is copy-and-paste. Since the code generated by an upper CASE tool is just nicely formatted yet plain text, it is easy to copy into the workspace of the lower CASE tool. In this way, the generated declarations and stubs can be reused and extended by application-specific code details.

As an example, consider the code generated with Enterprise Architect as shown in the figures 6-3 and 6-5. The programmer would copy this code from the EA's source-code pane and paste it onto the source-code pane of NetBeans. Since the result in NetBeans looks exactly the same as the source in EA, we refrain from showing the same in yet another figure.

Subsequently, the actual programming can start: The programmer will write Java code in NetBeans to implement, for example, the two "InvoiceItem" constructors:

```
public InvoiceItem(Advertisement advertisement,
                   Edition edition, price double) {...}
public InvoiceItem(Publication publication) {...}
```

The programmer might delete the generated parameterless constructor and then think about an efficient way to obtain the advertisement's price within the generated "getPrice" method, substituting the line "return 0" by a more meaningful result:

```
public double getPrice(){
   return 0;
}
```

6.2.3 Program Libraries and APIs

Using already existing code in the implementation stage is obviously a good way to reduce the implementation effort. Prefabricated modules are available in *program libraries* (also called module, class or code libraries). In fact, the total code of a typical software system today consists mainly of prefabricated modules. Only a very small percentage of the code is actually written by a programmer or generated by a CASE tool. The major part of the system is composed of existing code that was imported from program libraries.

Mathematical and statistical libraries

Program libraries have been created since computing began and contain reusable modules for many purposes. Application-oriented libraries have become particularly popular in mathematics and statistics. For example, the NAG Fortran libraries (http://www.nag.co.uk), the Matlab libraries (http://www.mathworks.com/products/matlab) and the IMSL libraries (http://www.vni.com/products/imsl) are very comprehensive program libraries containing thousands of subroutines for numerical and statistical problems. Other examples are low-level programming functions such as computer arithmetics, input/output, and operating-system functions made available to application programs through libraries.

Language extensions through libraries

In the development of programming languages, it became common to make languages extensible, or in fact to extend the language from the beginning on, with the help of program libraries. This means that a large part of the functionality is not provided through the language core but through extensions via libraries. Most functionality is thus not invoked through direct program commands but through library calls.

Program libraries are accessed through interfaces. Early libraries (e.g. numerical libraries) typically provided subroutines and functions as interfaces the programmer could invoke. Nowadays, especially in object-oriented languages, the interfaces are called *APIs (application programming interfaces)*.

Java APIs

A very typical language extensible through program libraries is Java. This language has only about 15 types of statements[§] and 50 keywords, presenting some desired properties or behavior the programmer may specify. However, thousands of predefined classes, along with the bare language, are available in class libraries! Programmers may use these classes in their programs. In fact, they are actually forced to do so because typical program features (such as reading input and writing output on a GUI) are only available from libraries.

A Java API is a class or interface specification

An API in Java is a specification of a class or an interface, describing how the class or the interface can be used. This means in particular that the attributes and the methods (including the constructors) are specified.

§ The weakener "about" is used because the number depends on what is counted as an individual statement type.

Figure 6-15 **Java API packages [Sun 2007]**

- **Java APIs**
  - **Java SE APIs**
    - Core APIs
      - java.awt - AWT (abstract windowing toolkit)
      - java.applet - creating and using applets
      - java.beans - developing JavaBeans
      - java.io - input and output
      - java.lang - fundamental classes of the Java language
      - java.math - performing arithmetics
      - java.net - implementing networking applications
      - java.security - security framework
      - java.sql, javax.sql - database access
      - java.text - handling text, dates, numbers and messages
      - java.util - utility classes
      - javax.naming - accessing naming services
      - javax.rmi - RMI (remote method invocation)
      - javax.swing - Swing: creating GUIs and more
      - …
    - Non-core APIs
      - com.sun.jdi - JDI (Java debug interface)
      - com.sun.security.auth - JAAS (Java authentication and authorization service)
      - com.sun.javadoc - inspecting source-level structure of programs and libraries
      - com.sun.mirror - mirror API
      - …
    - XML & web services
      - JAX-WS - Java API for XML-based web services
      - JAXP - Java API for XML processing
      - JAXB - Java API for XML binding
      - …
    - Other APIs
      - …
  - **Java EE APIs**
    - java.elb - EJB (Enterprise JavaBeans)
    - javax.faces - JavaServer Faces
    - javax.jms - Java messaging service
    - javax.jws - Java web services
    - javax.mail - modeling a mail system
    - javax.persistence - Java persistence
    - javax.servlet - Java servlets
    - javax.transaction - managing distributed transactions
    - javax.xml - contains several packages for XML
    - …
  - **Jave ME APIs**
    - …
  - **APIs for other technologies**
    - …

The meaning of the term "interface" in Java is different from the general meaning in software engineering. A Java interface is a reference type, similar to a class, that does not contain method bodies. Interfaces cannot be instantiated. They can only be implemented by classes or extended by other inter-faces [Zakhour 2006].

Java APIs are organized into packages containing related classes and interfaces. Packages can be nested, i.e. a package can be a part of another package which in turn can belong to yet another package etc. Therefore an identifier of a class member accessed through an API can contain a long path, for example:

```
javax.swing.border.LineBorder.createBlackLineBorder()
```

This is an invocation of the method "createBlackLineBorder" of the "LineBorder" class contained in the "border" package. The "border package is part of the "swing" package that is contained in "javax". (In practice, the developer would probably import the whole "border" package or the "LineBorder" class to avoid typing the long path name.)

Java APIs are available for all editions of the Java platform – Java SE (standard), Java EE (enterprise), Java ME (micro) and for a number of software technologies around, such as web services and e-mail.

Figure 6-15 summarizes the overall structure of the Java APIs and lists a subset of the available top-level packages. Note that the names are *package names* and not class names. The available prebuilt classes are contained inside the packages or in nested packages inside the outer packages.

The Java SE APIs are divided into core, non-core, XML & web services and other APIs. In the Java enterprise edition, more APIs are provided, including the EJB (Enterprise JavaBeans) APIs which are at the core of professional business applications today.

6.3 Testing

It is intentional that this section has a rather unspectacular title ("testing"). Other authors choose titles such as "software quality assurance", "software validation" or "software verification" for similar contents.

Although we will discuss different approaches as well, at the end it all comes down to testing.

6.3.1 Establishing Trust: Validation and Verification

Wouldn't it be nice if we could assure that a program is free of errors? A dream of all software engineers – yet impossible to reach for most non-trivial programs. It is a well-known fact in software engineering that software contains errors, just as other products contain errors. The only difference is that with tangible products the errors are called defects, not errors.

Before discussing efforts to make software correct, we first have to look at three closely related terms: software quality assurance, software validation and software verification.

Software quality assurance (SQA)

Software quality assurance (SQA) comprises all activities required to make sure that a software product meets certain quality objectives, in particular non-functional requirements such as maintainability, reliability, robustness, user-friendliness and understandability.

Software reliability

As a general term, SQA comprises all software quality attributes. However, in a narrower sense, the most relevant quality attribute here is *reliability*. As a property of a technical system, reliability is often defined with regard to time: What is the probability that the system will not fail to work as intended within a given time interval? Or: what is the mean time between failure (MTBF)? Quantitative figures such as these ones are important when *software metrics* [Ebert 2005] are used to in fact measure software quality.

Software validation

Software validation focuses on the external view of the system, especially with respect to the functional requirements (cf. section 5.1.1): does the system really do what the stakeholders want it to do? Barry Boehm gave the following definition of software validation: "To establish the fitness or worth of a software product for its operational mission" [Boehm 1981, p. 37]. He informally translated this to:

"Are we building the right product?"

Software verification

Software verification, on the other hand, aims to make sure that the software works as it should, according to its design specification. In Boehm's definition, the goal is: "To establish the truth of correspondence between a software product and its specification" [Boehm 1981, p. 37]. His informal translation of this is:

"Are we building the product right?"

Formal verification

The term verification is often used with a very specific, narrow meaning – in the sense of *formal verification* (also called program verification). This means using formal methods in order to *prove* that a program is correct. Formal methods require a formal specification of the software, usually with the help of a mathematical representation. Based on such a representation, a number of formal approaches (e.g. mathematical logic) can be applied to prove that the program does exactly what its formal specification states.

Generations of computer science students have been tortured learning formal verification methods and applying them to toy problems (such as stacks and queues). Real-world problems and realistically-sized software systems remained beyond the scope of what could reasonably be handled with such methods. Therefore, formal verification is not considered in this book.

Software developers try to produce programs that are free of errors, striving for correctness. But what exactly is an "error", and when is software "correct"?

What is an error?

Glen Myers introduced this question in 1976 in his seminal book on software reliability – still a mandatory text in many computer science courses today – with the following anecdote: "The Ballistic Missile Early Warning System is supposed to monitor objects moving towards the United States, and, if the object is unidentified, to initiate a sequence of defensive procedures ... An early version of this system mistook the rising moon for a missile heading over the northern hemisphere. Is this an error?" [Myers 1976, p. 4].

What is the answer to this question? – It depends. If the specification of the system stated what was quoted above, perhaps in a more formal way, then interpreting the moon as a hostile missile was correct – with respect to the specification. However, any reasonable person would probably say that this is an error, because taking "defensive procedures" against the moon does not make much sense.

Obviously the notion of an error depends on the standpoint of the observer. The above example illustrates that there are at least two interpretations of what may represent an error:

– From a *software-technical* perspective, any deviation of the program's behavior from its specification can be considered an error.

– From a *user-oriented* perspective, an error occurs when the program does not do what the user can reasonably expect it to do.

Ensuring trust in software reliability

Neither from the software-technical nor from the user-oriented point of view is it really possible to guarantee that a program is free of errors. A formal proof of correctness is beyond what is feasible today. Therefore other ways have to be gone in order to assure a certain level of trust in the program. As we cannot reach faultlessness, methods to ensure sufficient trustworthiness have to be applied. The aim of these methods is to credibly assure that the performance of the program will be satisfactory, in particular that the program contains only a small number of errors, making failures reasonably unlikely.

Approaches to find errors and to improve the reliability of software can be put into two categories: manual (or document-based) and automated (or tool-supported) approaches.

Document-based approaches

Manual approaches are based on paper documents such as requirements specifications, analysis and design models, use cases, interface descriptions (e.g. methods' signatures), pseudocode and source code. Document-based methods have names such as "design review", "code inspection", "walk-through", "structured walk-through" and "technical review".

These methods are accompanied by organizationals concepts, regarding both the process and the composition of the group performing the inspection or review. For example, in a code review, the author of the code (programmer) leads the group through the code, participants (readers and inspectors) ask questions and make comments about possible errors, and a moderator or chairman facilitates the inspection process.

"IEEE standard for software reviews"

Document-based approaches are also called *static* approaches, because they do not require software to be executed (which would be considered "dynamic"). These approaches have been available and used in practice for many years. The IEEE even defined a standard for inspections, reviews and walk-throughs under the code IEEE 1028-1997 ("IEEE standard for software reviews") [IEEE 1998b].

Automated approaches

Automated approaches are applied to *software*, which implies the use of tools. Therefore they are also called tool-supported approaches. A typical example is a software component being examined with the help of a compiler and a debugger. Software testing is another name for tool-supported approaches in order to find errors and improve program reliability.

6.3.2 Testing Principles and Test-case Design

Software testing has been called a "destructive" activity because its goal is to show that errors are there, not to prove correctness! If anything can be "proved" by testing, then it is that the software is faulty. One of the most popular quotes in the testing literature goes back to Edsger Dijkstra, the father of structured programming: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra 1970, p. 7]

Software testers are "destructive"

In this sense, testers need to have a "destructive" attitude: find as many errors as possible! In order to succeed in doing so, adequate test cases have to be designed and applied. Test-case design depends on the chosen testing strategy. A common distinction of testing strategies is functional (black-box) versus structural (white-box) testing, although more than these two strategies are available.

Functional testing (black-box testing)

A program is a black box

Functional testing focuses on the functionality of the software as it is perceived from "outside" the piece of software under testing. The software is considered a *black box*, accessed only through its interfaces. Since the functionality is tested and not the working of internal program mechanisms, the tester does not look inside the program but observes its external behavior. Functional testing means testing against a specification. Test cases are designed in such a way that valid and invalid input and expected output are specified.

Partition testing

Since it is clearly not feasible to test all possible input data constellations, an approach to functional testing is to partition the inputs into groups which are expected to cause the same program behavior *(partition testing)*. For example, if a test case is based on input from a 20-item listbox and the test is run with one particular item selection from the list, then it can be assumed that the program will behave in the same way when any other item from the list is selected.

As another example for partition testing, consider an inventory replenishment module dealing with on-hand stock. Provided that the values for on-hand stock this module receives are numerical, they can possibly be

– greater than the reorder level,
– less than or equal to the reorder level,
– negative.

The possible input data in this example would be divided into three partitions. It is assumed that testing the module with one value out of a partition (e.g. "+34 units on stock") is sufficient, because it can be expected that testing the module with another value (e.g. "+49 units on stock") will not create any other behavior of the program. It should be mentioned that this assumption is certainly reasonable, but it may also be wrong.

Equivalence classes (equivalence partitions)

Common partitioning strategies are partitioning into valid and invalid inputs, and into correct and incorrect outputs. Partitions of program inputs or program outputs are also called *equivalence classes* or *equivalence partitions*.

Boundary-value analysis

Supplementary to forming partitions, it can be helpful to look at the values at the boundaries of the partitions, and at the boundaries of value ranges in general *(boundary-value analysis)*. Another typical value to check is zero. In the above example, suppose the reorder-level quantity is 25. Typical boundary values of on-hand stock to test would then be 26, 25, 24, 1, 0 and -1.

Developing test cases

Test cases for black-box testing should be developed by persons who understand the interface mechanisms of the software under consideration (e.g. how to invoke it) but who do not know the internal structure of the software. In practice, IT organizations usually have specialized testing departments, testing groups or individuals who develop test cases for black-box tests and run them.

Structural testing (white-box testing)

Looking inside the "box"

The opposite of black-box testing is *white-box testing*. This means looking inside the "box", i.e. looking at the program code and the program structure. Therefore it is also called *structural testing*. White-box testing deals with the internal program logic. How detailed the logic is examined depends on the chosen strategy. Several approaches have been proposed:

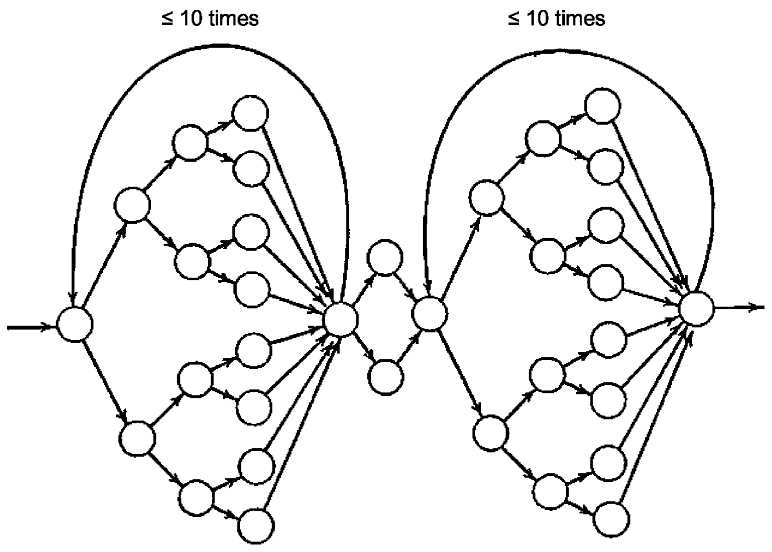
– *Statement coverage:* Execute and test every single statement of the program at least once.

- *Condition (or branch) coverage:* Execute and test every point in the program where a branching decision has to be made at least once.
- *Path coverage:* Execute and test every path through the program at least once.
- *Invocation coverage:* Execute and test every subprogram invocation (entry point and exit point) at least once.

10^{18} paths through the program

For non-trivial programs, creating test cases according to these approaches can be extremely difficult or even impossible. How many test cases would be needed to check every path through the small program outlined in figure 6-16? The program graph contains 2 loops with at most 10 iterations each and 14 branching decisions, yielding approximately 10^{18} paths through the program. How would anyone create 10^{18} test cases and check whether they are correctly executed? 10^{18} is 10 quintillions (i.e. a 1 with 10 zeroes).

Figure 6-16 **Flow graph of a program with loops and branches**



Condition coverage

Although path testing and other white-box testing approaches have been around in software-engineering textbooks for decades, and still are, their utility is questionable. Complete path coverage usually cannot be achieved. *Condition coverage* is more reasonable as it reduces the number of program paths significantly. Including the looping conditions of figure 6-16 in the counting, the number of paths to test for condition coverage is 162 – still a substantial number, but more manageable. Test-case automation with tools (see below) can help to create the test cases.

For practical testing, white-box approaches make more sense when they are not applied ex post (i.e. to a completed piece of software) but as an ongoing effort. This means testing in an incremental way, in parallel to the writing of the program code. This aspect of white-box testing will be discussed further below (cf. regression testing, test-driven development).

White-box testing is a supplement to black-box testing. When defects have been elaborated in the black-box tests, then white-box testing can help to locate the causes, i.e. erroneous statements in the program code.

Test cases for white-box testing are developed by persons who know the internal structure and working of the software. These are usually the programmers who developed the piece of software under consideration.

Gray-box testing

Merging functional and structural testing

Black-box testing is done from a completely external view – without knowledge of the internals of the piece of software under testing. White-box testing is the exact opposite: From knowing exactly what the program code is like, the tests are built. *Gray-box testing* is a combination of both, trying to unite the advantages of both approaches. In gray-box testing, test cases are created as in black-box testing, but making use of knowledge of the internal structure of the test object at the same time. Functional and structural aspects of testing are merged.

"Test first" principle

Some widespread approaches and also a number of individual approaches fall into the category of gray-box testing. In the section on XP (extreme programming), we mentioned the *"test first" principle* as one of the XP cornerstones (cf. section 4.4.1). This principle is realized through gray-box testing, within the more general framework of TDD (test-driven development, cf. section 6.3.3 below).

Rapid testing

Another gray-box testing approach called *rapid testing* focuses on finding the most severe errors first. This approach was motivated by the fact that it is impossible to test everything – neither through white-box nor through black-box testing. On the other hand, not all errors are equally grave with respect to the utility of the software.

There are many types of errors, for example:

How severe is an error?

- » Fatal errors – make it impossible to use the software as intended.
- » Severe errors – allow the use of the software, but prevent that it works properly.
- » Severe errors with work-arounds – prevent that the software works properly, but the error can be bypassed through additional measures.
- » Other errors – impairing the use or the look of the software, but not really severe.

Instead of writing test cases for black-box or for white-box testing first, rapid testing starts with a "mission":

"Find the most severe errors first."

Rapid testing doesn't mean 'not thorough'

The term "rapid testing" reminds of rapid application development (RAD), an approach to developing application systems quickly. The negative side of RAD is that if it is not done in a systematic way, it can result in "quick-and-dirty" development. From this perspective, James Bach, an advocate of rapid testing, stresses that "... rapid testing doesn't mean 'not thorough', it means 'as thorough as is reasonable and required, given the constraints on your time.' A good rapid tester is a skilled practitioner who can test productively under a wider variety of conditions than conventionally trained (or untrained) testers." [Bach 2006]

Exploratory testing

Exploratory testing is a related approach, stressing that not everything in testing can be prescribed beforehand, through writing and subsequently executing test cases. "Exploratory testing is simultaneous learning, test design, and test execution" [Bach 2003]. This means that the tester actively controls the design of the tests as these tests are performed. The tester learns more about the software and uses information gained while testing for the design of new and better tests.

Exploratory testing can be seen as the opposite of traditional, planned testing. The latter form is also called *(pre-) scripted testing* because the tests are planned before and described in a written specification (at least in theory). Exploratory testing, on the other hand, takes the standpoint that the tester learns in the process and therefore the testing evolves.

Regression testing

Software lives and evolves. During its initial development, the state of the software changes permanently. Throughout the lifetime of the software, more changes happen.

Things that worked before don't work any more

A recurring problem with evolving software is that things that worked before suddenly do not work any more. This happens both in the micro iterations, within a development cycle, and in the long-term system evolution, from one release to the next one. When a new version of the software is created, based on an existing and running version, special attention has to be devoted to ensuring that properly functioning features of the previous version continue to work in the new version. For the developer, this means that the tests performed against the old version must still run, with the same results, against the new version.

Going back in the test history

Repeating tests that were performed with earlier versions of the software is called *regression testing.* Here, regression means going back in the test history. Taking into consideration that the same system may have undergone a number of versions (or releases), the tests for each of the earlier versions need to be successfully rerun. The new version can contain new errors, but it also occurs that previously fixed errors re-emerge. This can be caused, for example, by the unforeseen effects of program changes ("spaghetti-bowl effect"). It can also happen when parts of the software are redesigned and reimplemented, and the developer again makes the same mistakes as before.

Figure 6-17 illustrates the idea of regression testing. Just as the software undergoes an evolution, through a number of versions, the regression test suite grows. The suite for the current version n comprises new tests created for this version, in particular for new features, but also tests that were run against the previous version n-1. These tests include regression tests of version n-2 etc.

Keeping track of all tests

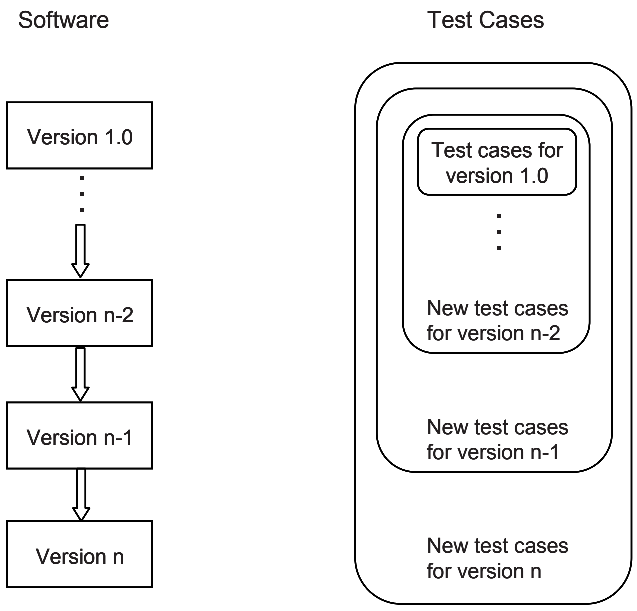
A major problem of regression testing is to keep track of all the tests that were created and run before. Especially when software has a sequence of releases, information regarding earlier tests often does not survive. Such information can be, for example, why certain test cases were developed, which problem they checked, and if they are still valid.

Keeping track of test cases across versions with manual procedures is cumbersome. Automated testing tools can help to record test cases and execute all earlier regression tests automatically. With the help of a tool, it is even possible to rerun all regression tests automatically at specified intervals, e.g. once a day (during development) or once a week, and record all errors.

Test specification and documentation

In a "disciplined" approach to software development (as opposed to an "agile" approach, cf. section 4.4.1), things are planned and documented before they are executed. With regard to testing, this means that tests are planned and test cases are developed before the testing starts. (The opposite is exploratory testing as mentioned above.) The result of the planning is a *test plan.* A formal description of the plan, laid down in a document, is called a *test specification*. Many IT organizations, and individual projects as well, have developed their own guidelines of what to document in a test specification and how.

Figure 6-17 **Evolution of software versions and regression test suite**


Software
Test Cases
Version 1.0
Version n-2
Version n-1
Version n
Test cases for version 1.0
New test cases for version n-2
New test cases for version n-1
New test cases for version n

Keeping track of all tests

A major problem of regression testing is to keep track of all the tests that were created and run before. Especially when software has a sequence of releases, information regarding earlier tests often does not survive. Such information can be, for example, why certain test cases were developed, which problem they checked, and if they are still valid.

Keeping track of test cases across versions with manual procedures is cumbersome. Automated testing tools can help to record test cases and execute all earlier regression tests automatically. With the help of a tool, it is even possible to rerun all regression tests automatically at specified intervals, e.g. once a day (during development) or once a week, and record all errors.

Test specification and documentation

IEEE 829-1998 standard for test documentation

A widely accepted, comprehensive standard for test documentation is the IEEE 829-1998 standard. This standard specifies the form and content of individual test documents. Its terminology is a little different insofar as the term "test plan" is used to describe the testing on a higher abstraction level, while the term "test specification" stands for refinements of the test plan regarding major activities.

IEEE 829-1998 covers a set of test documents for test planning, test specification and test reporting [IEEE 1998a]. Test specification documents are defined for the description of the test design, the test cases and the test procedure. For test reporting, four document types are recommended to describe what happens during execution of the tests. Figure 6-18 quotes the description of the test documents from the standard.

The IEEE standard describes eight types of documents. In practice, few organizations really create all these documents. A typical *test specification* will consist of one document, or of two documents (with test-case descriptions in a separate document), and the reporting is also condensed into one document.

"Software bureaucracy"?

An outline of a typical test-specification document is given in figure 6-19. This outline shows that in a "disciplined" testing approach a large number of details are specified. (Advocates of agile development and exploratory testing call this "software bureaucracy.") While the documentation overhead might seem frightening at first glance, it becomes more and more important as time passes. Suppose three years later, a new tester has to find an error detected by a regression test. This person will appreciate very much any piece of information that was recorded three years ago, including the telephone numbers of the people listed in the first section (and hope that any one of them will still be working for the company).

Example: 36 test cases

The number of test cases that have to be planned for a unit to test depends on the testing strategy and on the complexity of the unit. In a black-box strategy, the number of test cases depends primarily on the possible inputs. Suppose we want to test the replenishment module mentioned above in the subsection on black-box testing. Under the assumption that an input to this module consists of two values, the stock on hand and the type of the product (A, B or C, according to an ABC classification), then we might want to test all combinations of:

– 9 values for stock on hand from equivalence classes and boundary values, e.g. 34, 5, -22; 26, 25, 24, 1, 0 and -1;

Figure 6-18 **Documents of IEEE standard 829-1998 [IEEE 1998a, p. iii]**

Test Plan

The test plan prescribes the scope, approach, resources, and schedule of the testing activities. It identifies the items to be tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with the plan.

The Specification

Test specification is covered by three document types:

- A test design specification refines the test approach and identifies the features to be covered by the design and its associated tests. It also identifies the test cases and test procedures, if any, required to accomplish the testing and specifies the feature pass/fail criteria.
- A test case specification documents the actual values used for input along with the anticipated outputs. A test case also identifies constraints on the test procedures resulting from use of that specific test case. Test cases are separated from test designs to allow for use in more than one design and to allow for reuse in other situations.
- A test procedure specification identifies all steps required to operate the system and exercise the specified test cases in order to implement the associated test design. Test procedures are separated from test design specifications as they are intended to be followed step by step and should not have extraneous detail.

The Reporting

Test reporting is covered by four document types:

- A test item transmittal report identifies the test items being transmitted for testing in the event that separate development and test groups are involved or in the event that a formal beginning of test execution is desired.
- A test log is used by the test team to record what occurred during test execution.
- A test incident report describes any event that occurs during the test execution which requires further investigation.
- A test summary report summarizes the testing activities associated with one or more test design specifications.

– 4 values for ABC classification, e.g. "A", "B", "C" plus one illegal value ("9").

This means that 36 combinations (9 times 4) are possible, and thus 36 test cases will be written. As in this example, the number of test cases is

Figure 6-19 **Outline of a test specification (example)**

Document Information

- Title and identifier of the document
- Project name and identifier
- Document type, summary, and purpose of the document (what is specified in this document?)
- Date, version and version history (who changed what when?)
- Document status (draft, final, ...), confidential/not confidential
- Author of the document, responsible or authorizing person, contact person
- Other relevant, related documents

General Information

- Short description of the project/software system
- Overall goals and objectives of the tests
- Scope (functionality/features/behavior to be tested and also, not to be tested)

Test Preparation

- Testing schedule
- Preconditions for the tests
- Test objects (what is to be tested?)
- Test strategy (black-box, white-box, path coverage etc.)
- Procedure to determine test cases
- Repository for test cases
- Termination criterion (when to stop testing)
- Procedure to create/adopt test data (manual, from database, with the help of a generator etc.)

Testing Environment

- Testing platform
- Testing and documention tools
- Test files and/or test databases
- Special hardware/software (e.g. test server), generators and other resources

Test Organization

- Persons (roles and names) involved in the testing and their responsibilites

Test case 1

Test-case ID
- Test object
- Test input (data) and preconditions for the test
- Test-case description, actions to be performed, test steps
- Expected result (output data or behavior)
- How and where to record defects

Test case 2

…

Test case n

…

Test Reporting Specification

- Test steps performed and defects noticed, for each test case
- Procedures for storing and evaluating test results
- Definition of a test log (chronological record of all tests and their results)

usually quite large. Therefore the test-cases are often described in a separate document, called a *test-case specification*.

Test automation

Automated tools for testing tasks

Testing is a lot of work and it takes a lot of time. In some software development projects, 50 % of the development effort go into testing. Therefore most organizations are moving from manual testing to automated testing, completing three major tasks:

– *Test-case execution:* If the test cases are already available, then an automated tool can run the tests and record the results (success, failure, details of the defects, state of the environment etc.).
– *Test-case design:* In addition to test case execution, tools can help to *design* the test cases – in a way that they can be executed automatically. This requires some sort of formal specification of the software behavior, which is often not easy to obtain. However, tools to specify test cases are available today.
– *Debugging:* This is the part of testing that is supported best by today's tools. Debugging support will be discussed and demonstrated below in section 6.3.5.

How much testing is enough?

Any non-trivial software system contains errors. After testing and debugging the system will contain fewer errors, but most likely the system will not be error-free. Testing costs a lot of money – time, manpower and computing resources – so the question arises: when to stop testing?

When to stop testing?

In practice, this question is often answered in a rather pragmatic way: when all defined test cases – manually defined and/or tool-generated – run without errors; when the tester feels that all typical inputs are processed satisfactorily (or when the tester knows which inputs to avoid because they make the program crash); or simply when the time scheduled for testing is over.

Not all errors are equally severe. It is obviously more important to remove severe errors than to remove insignificant errors. The above mentioned *rapid testing* approach started from the mission: "Find the most severe errors first." While it makes sense, abstractly speaking, to start with the most severe errors and then to proceed to the less severe ones, this does not answer the questions: When should testing stop? What errors are severe? What errors are insignificant? Obviously the answer depends on how much damage an error can cause and/or how much value the stakeholders obtain from removing the error.

Value-based software engineering (VBSE)

An approach to relate the efforts made in the software life cycle to the value created by these efforts is *value-based software engineering (VBSE)* [Biffl 2006]. It is about making such decisions in software engineering that the value of the delivered system will enhanced.

Value-based testing

As a part of VBSE, *value-based testing* relates the adequate amount of testing to the risks that have already been addressed in testing and to the risks that are still open. This approach assumes that risks were identified before, in a risk analysis within the requirements engineering stage, and test cases have been defined to address these risks.

A Cocomo II and Coqualmo based method for testing planning

Huang and Boehm present a quantitative method to determine "how much testing is enough", based on the *Cocomo II* cost-estimation model (cf. section 2.4.3) and the *Coqualmo* quality-estimation model[§]. They show that there is an optimal point when to stop testing [Huang 2006, pp. 93-94]. This point is at the minimum of a risk-exposure (RE) curve constructed from the probability of loss P(L) and the size of the loss S(L), as shown in figure 6-20:

$$RE = P(L) * S(L)$$

Loss can refer to either financial loss or loss of reputation and therefore diminished future prospects. The fewer the defects, the lower the probability of loss. The abscissa represents the needed investment in software quality (SQ), or more simply, the effort for testing the system.

Stop testing at the "sweet spot"

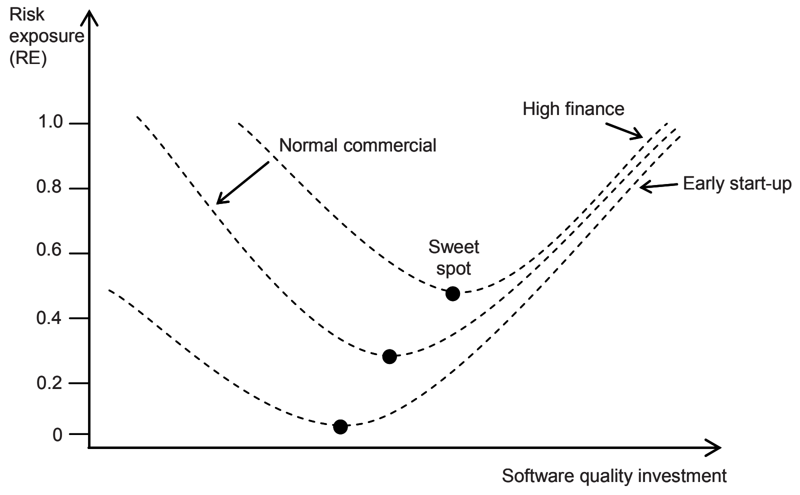
The minimum ("sweet spot") is different for different types of software systems. In figure 6-20, three curves are shown. The curve on top, with the "sweet spot" at the right-most side, is an example of a system with very high reliability requirements. (The authors describe it as a "high-finance business case ... representing very high-volume time-sensitive cash flows" [Huang 2006, p. 93]). It needs the most testing effort. The curve below is for a less critical system, a "normal commercial" system, and the lowest curve is for a system where some errors may even be tolerated ("an early start-up representing relatively defect-tolerant early adopters" [Huang 2006, p. 93]).

Value-based testing is superior to conventional testing

Huang and Boehm show also that value-based testing is superior to conventional (value-neutral) testing, e.g. testing with automated test generators, path testing or testing with unprioritized requirements. Value-based testing produces more business value per dollar invested than conventional testing.

§ Online information on Coqualmo is available at http://sunset.usc.edu/research/coqualmo/coqualmo_main.html (accessed July 17, 2007).

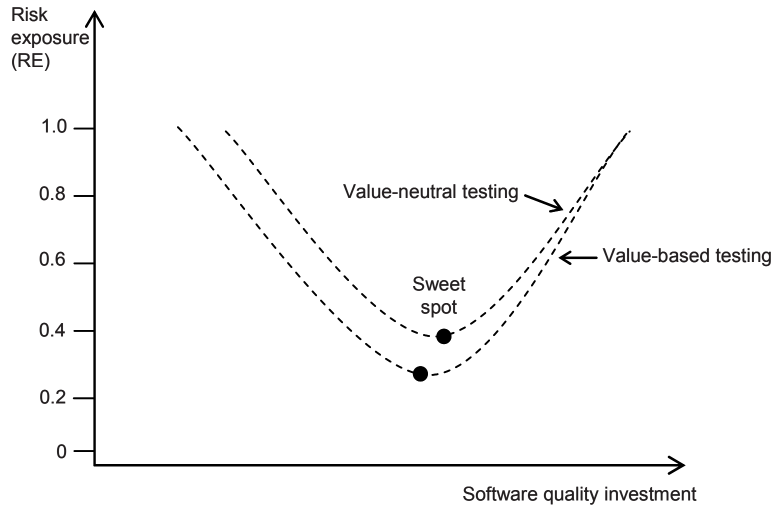
Figure 6-20 **Risk exposure vs. quality investment [Huang 2006, p. 93]**



The authors exemplify their findings with a comparison of value-neutral testing versus value-based testing of the "high-finance" system mentioned above. As figure 6-21 illustrates, the "sweet spot" for value-neutral testing is higher and more to the right than for value-based testing. This means that value-based testing costs less and has a lower risk exposure than value-neutral testing.

The approach of Huang and Boehm is plausible. However, a prerequisite is that risks are actually addressed and explicitly quantified as it is the case in the Cocomo II and Coqualmo models. Organizations using these models can apply value-based testing. Others can at least adopt the general insight from VBSE, that the most severe errors to remove first are those that can cause the most severe damage to the organization's business.

Figure 6-21 **Risk exposure of value-based vs. value-neutral testing**[§]



6.3.3 Test-driven Development (TDD)

"Test first" principle

The "test first" principle has gained its popularity as one of the practices of XP (extreme programming). *Test-first development (TFD)* is an implementation of this principle, employed inside the XP community as well as outside. TFD as a general approach to testing and programming reverses the sequence of activities: write tests first and code afterwards.

This approach implies incremental development: Before the next small piece of the program code is written, the programmer thinks of what capability this code should add to the program and what the result should be. Prior to writing the code, the programmer writes a test invoking the capability and examining the results provided by it.

§ Huang 2006, p. 94.

Write tests first and code afterwards

Running the test immediately would yield a program failure, because the capability is not yet implemented. The necessary code is written with the goal in mind to make the test pass. Tests that had been run against the program before the new capability was added obviously must still pass, so all are repeated.

Refactoring

Since the program grows in small increments, this approach bears the risk that an initially clear design becomes blurred. Therefore, the program structure is reexamined and improved through *refactoring*. Refactoring is a disciplined approach to restructuring code, making small changes to improve the program structure without changing the behavioral semantics, i.e. neither removing nor adding behavior [Ambler 2007, p. 39].

Designing and writing a test, implementing code so that the test passes, and improving the program structure via refactoring are the essential activities in each micro-iteration. The next increment of the code is created in the same way. The program grows and its structure evolves in micro-iterations through refactoring.

Test-driven development (TDD) is a combination of test-first development (TFD) and refactoring [Ambler 2007, p. 37]. The following definition is based on Jeffries and Melnik's characterization of TDD and reflected in figure 6-22§ [Jeffries 2007, p. 24-25]:

Definition: test-driven development

> *Test-driven development* is a discipline of design and programming where every line of new code is written in response to a test the programmer writes just before coding and where the entire code is reviewed after each code increment.

TDD helps to build up a regression test suite

Test-driven development has gained much popularity among programmers mainly due to two reasons: TDD helps to ensure that things that worked before still work when the program is changed, and it helps to build up a regression test suite in a quite natural way. Regression tests are the major mechanisms to ensure continuous functioning of the program.

Due to a weak design and obscured or opaque program structures, large programs are sometimes very hard to implement, modify or extend. This is particularly true for legacy systems where the only infor-

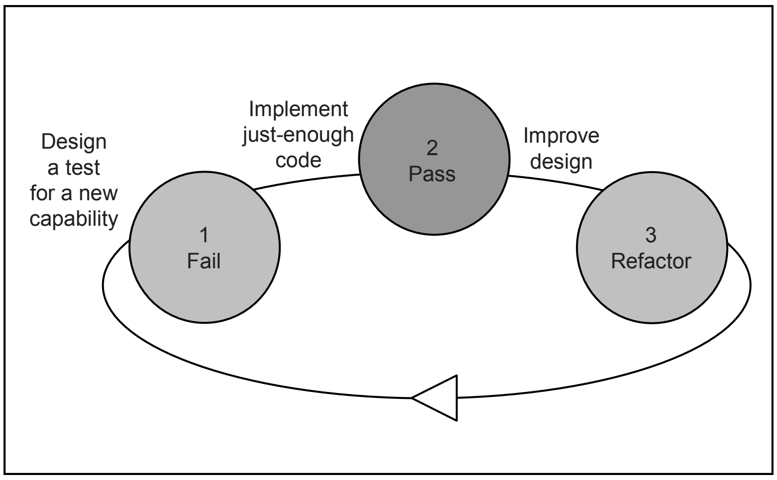
§ Jeffries and Melnik actually use the term "design" instead of "structure" [Jeffries 2007, p. 24-25]. We prefer to speak of a program "structure" in the figure, because the term "design" in software engineering is normally used for creative activities on a higher abstraction level than coding.

mation the programmer has is the source code (a "spaghetti bowl" of code). Making changes to such a system can be nightmare.

"Don't touch a working program!"

TDD is a considerable step forward compared to the state-of-the-art of implementation and testing. Many developers and maintenance programmers of large systems are afraid of unforeseen effects when they have to add new features or make other changes to a working program (including such actions during the development process). While this state of affairs of practical software engineering may be lamentable, it is the real world. TDD can help to raise the programmer's confidence in the program – and in his/her own actions[§].

Figure 6-22 **TDD micro-iterations [Jeffries 2007, p. 25]**


Design a test for a new capability
Implement just-enough code
Improve design
1 Fail
2 Pass
3 Refactor

Errors show immediately

Writing comprehensive program code for new features first and testing afterwards often does not work because when errors occur they are difficult to find and debug. In an extremely incremental approach such as TDD where almost every new line is immediately tested, errors show more or less immediately. A typical duration of a test-code-pass cycle as in figure 6-22 is just a few minutes (excluding refactoring). Regression tests are also run every few minutes. In this way, any new

§ This lamentable situation was mirrored in a paper entitled "TDD: The art of fearless programming" – the editors' introduction to a special issue of the IEEE Software magazine devoted to test-driven development [Jeffries 2007].

problem created by an additional or modified line of code becomes visible immediately.

The larger the program and the more tests in the regression suite, the longer the regression testing takes. Therefore the complete test suite may not be run after every increment but at longer time intervals.

Martin reports about a 45,000 lines-of-code Java program developed in the test-driven manner where running the complete regression test suite took 30 seconds [Martin 2007, p. 34]. Since this was considered too long a waiting time, the complete suite was run only about every 30 minutes. Tests for the immediate environment of the current increment took only 1 to 2 seconds. This means that new errors were discovered after a few minutes or at most after half an hour.

The very short cycles of TDD are advanced by keeping things as simple and short as possible. Martin coins this rule into two of his "three laws of TDD" as shown in figure 6-23.

Figure 6-23 "The three laws of TDD" [Martin 2007, p. 32]

The Three Laws of TDD
1. You may not write production code unless you've first written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail.
3. You may not write more production code than is sufficient to make the failing test pass.

TDD is also used for GUI and database development

While TDD is mainly used for the development of code pieces – on the module or unit level – it has been applied to special aspects of information systems development as well, such as developing GUIs [Ruiz 2007] and databases (TDDD – test-driven database development [Ambler 2007]). TDD is also used on higher testing levels such as acceptance testing (cf. section 6.3.4).

TDD has obvious benefits but also drawbacks. Gains in productivity have been reported in many studies, but the results are sometimes controversial, which does not allow generalization. A summary of such studies is presented by Jeffries and Melnik [Jeffries 2007, pp. 27-29]. Most of the listed studies report significant gains in productivity and software quality, but others show no or even negative effects. Substantially lower error rates are a frequently reported effect of TDD.

TDD conflicts with conventional implementation and testing

Test-driven development conflicts with conventional implementation and testing methods which are practiced in many organizations – just as agile development and extreme programming conflict with conventional software development. Organizational structures can impede TDD. The IT management can be in favor of "disciplined" approaches to software development (cf. section 4.4.1). If software quality assurance (SQA) or software quality management (SQM) is installed as an organizational unit, then this unit expects larger pieces of written code (modules, packages, subsystems etc.) to test and not individual statements.

TDD is hard to learn

A disadvantage of TDD is that it is hard to learn. For traditional programmers, it is difficult to figure out how to create effective tests, and to do so ahead of having even code to test. On the other hand, further up the learning curve, writing tests becomes much easier, and programmers who have experienced TDD for a while prefer it over traditional implementation and testing [Crispin 2006, p. 71].

An inherent drawback of incremental development is that the final program structure is likely to be less "clean" than a structure that was designed and implemented top-down. Through refactoring, the structure is continuously improved, but this is also to some degree an incremental improvement. Radical refactoring – i.e. structural changes – collides with the above mentioned fear of breaking something that used to work.

"Fearless programming"

Why do programmers not clean up code? Martin answers this question with a quote reflecting a common attitude among software developers: "If it ain't broke, don't fix it!" [Martin 2007, p. 33] Test-driven development, on the other hand, lets programmers improve code without the fear of breaking something, because any small mistake will be detected immediately.

6.3.4 Testing Levels and Scope

Testing comprises activities with different scopes, beginning with the testing of single code units and ending with the complete software system ready to go into operation. The levels on which tests are performed are:

– Module testing
– Integration testing
– System testing

– Installation testing
– Interoperability testing (system integration testing)
– Acceptance testing

Module testing

The goal of *module testing* (also called component testing or unit testing) is to ensure proper functioning of a piece of software such as a class or a method. Under the name *unit testing*, this type of testing has received a lot of attention, in particular within the TDD community, not least owing to that unit testing is well supported by test tools. However, *any* testing sequence, not only in TDD, starts with the testing of small software units (modules, components).

The discussion about black-box versus white-box testing in section 6.3.2 was mostly a discussion about module testing. Various approaches to module testing exist, including data-flow, fault and usage based testing as shown in figure 6-24. This figure is derived from a survey by Juristo et al. who investigated experimental results of different types of unit testing [Juristo 2006, pp. 74-75].

Specification-based techniques are basically black-box approaches, and code-based techniques are white-box approaches. *Data-flow based* approaches among the latter ones address the path from a variable definition and its uses in the program, i.e., the code between the definition and the use of the variable is executed. *Reference-model based* techniques derive the test cases from a graphical representation of the program in a flow graph or call graph. A number of diversifications of control-flow, data-flow and mutation based techniques listed in Juristo et al.'s survey have been omitted in the figure.

The authors discuss some results of the studies they examined, pointing out that the results have to be considered with caution as far as generalization is concerned. Some interesting findings include the following [Juristo 2006, p. 73]:

Results of the study

– Specification-based techniques are usually more effective than code-based techniques that use criteria with weak coverage levels.
– Boundary-value analysis was found more effective than statement coverage but takes longer and needs experienced testers.
– Condition coverage takes longer than boundary-value analysis, whereas its effectiveness is similar.

Figure 6-24 **Techniques for module testing [Juristo 2006, pp. 74-75]**

Techniques for Test-set Generation

Based on tester's intuition and experience			
	Ad hoc testing		*- no specific testing guidelines*
	Exploratory testing		*- test cases are dynamically modified based on experience*
Specification based			
	Equivalence partitioning		*- cf. section 6.3.2*
	Boundary-value analysis		*- cf. section 6.3.2*
	Decision table/cause-effect graphing		
	Finite-state-machine based		
	Derived from formal specifications		
	Random testing		
Code based			
	Control-flow based techniques		
		Statement coverage	*- cf. section 6.3.2*
		Decision (branch) coverage	*- cf. section 6.3.2*
		⋮	
		Path coverage	*- cf. section 6.3.2*
	Data-flow based techniques		
		All definitions	*- test cases cover each definition of each variable for at least one use of the variable*
		All uses	*- test cases ensure that there is at least one path of each variable definition to each use of the definition*
		All c-uses	*- same for: … to each use of the variable in a computation*
		⋮	
		All du-paths	*- test cases must execute all possible paths of each definition of each variable to each use of the definition*
		All dus	*- same for: … all possible executable paths ...*
	Reference-model based techniques		
		Flow graphs	*- test cases are derived from a graphical representation of the program in a flow graph or call graph*
		Call graphs	
Fault based			
	Error guessing		*- test cases are derived from knowledge about typical faults*
	Mutation testing		*- test cases must cover all (or some) program mutants*
Usage based			
	Operational profile		*- test cases are generated according to their probability of occuring in an actual operation*
	Reliability-engineered testing		*- tests are designed according to reliability objectives, expected use, and criticality of different functions*
Fault seeding			*- faults are artificially introduced in the program to evaluate test-set quality*

Unit-testing tools

JUnit, SUnit, CUnit etc.

The growing popularity of test-driven development is partly due to the fact that powerful testing tools are available. These tools are known under the name *xUnit*, where x stands for a programming-language initial: JUnit is the tool for Java, SUnit for Smalltalk, CUnit for C, CppUnit for C++, NUnit for the .NET languages etc.

Testing code is separate from implementation code

A common feature of this tool family is that testing code is separated from the actual implementation code. The tester writes testing code with xUnit annotations regarding the code unit to be tested. xUnit tools process the annotations and create a compact testing protocol which is outside the unit under testing. This is a definite advantage over conventional module testing where testing code is usually embedded in the actual implementation code.

Erich Gamma, Kent Beck

Although *JUnit* is the most popular tool from the xUnit family, the origin of the tool family was a pattern and a framework written by Kent Beck for Smalltalk (SUnit). Ported to Java by Erich Gamma and Kent Beck, JUnit is an open-source framework to write and run repeatable unit tests. Its features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test runners for running tests

"JUnit cookbook"

Beck and Gamma encourage the use of JUnit due to the obvious advantages of automation: The test runs as such can be automated; many tests can be run at the same time; and the test results can be interpreted automatically, without human interference. In their "JUnit cookbook", Beck and Gamma introduce JUnit in the following way [Beck 2007]:

> "JUnit tests do not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do:
>
> 1. Annotate a method with @`org.junit.Test`.
> 2. When you want to check a value, import `org.junit.Assert`.* statically, call `assertTrue()` and pass a boolean that is true if the test succeeds.
>
> For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, write:

```
@Test public void simpleAdd() {
   Money m12CHF = new Money(12, "CHF");
   Money m14CHF = new Money(14, "CHF");
   Money expected = new Money(26, "CHF");
   Money result = m12CHF.add(m14CHF);
   assertTrue(expected.equals(result));
}
```

If you want to write a test similar to one you have already written, write a Fixture instead."

In this example, the "assertTrue" method checks whether the "result" (the value returned by the tested "add" method) is equal to what was "expected", and returns true or false.

Test fixtures

A *test fixture* is a set of test data (objects) that can be used for more than one test. Using such a fixture avoids duplicating the code necessary to initialize the common test objects. In many cases, setting up test objects takes more time than setting up the test as such. Once the fixture is defined, it can be used for any number of test cases. A fixture is invoked with an "@org.junit.Before" annotated method in which the instance variables for the common objects have to be initialized.

Continuing the above example, the "@Before" method "setUp" could be used to define a set of test data, such as 12 CHF, 14 CHF and 28 USD, that is available for all test cases [Beck 2007]:

Figure 6-25 **Java class with JUnit testing annotations**

```
public class MoneyTest {
   private Money f12CHF;
   private Money f14CHF;
   private Money f28USD;

  @Test public void simpleAdd() {
    ...
  }

  @Test public void simpleSub() {
    ...
  }

  @Before public void setUp() {
    ...
  }

  @After public void cleanUp() {
    ...
  }
}
```

```
@Before public void setUp() {
   f12CHF = new Money(12, "CHF");
   f14CHF = new Money(14, "CHF");
   f28USD = new Money(28, "USD");
}
```

Running the tests

For *running the tests* and collecting the results, JUnit provides tools to define the test suite and to display the results. Suppose we created another test case "simpleSub" checking a subtraction, then the class to test might look like the one outlined in figure 6-25. A JUnit method tagged with "@After" is automatically invoked after each test is run (just as an "@Before" method is executed before each test is run).

The "MoneyTest" class will be invoked for execution by a so-called *runner*, either from the console, from inside an IDE or from another Java program. For example, an invocation from the console is done by running the "JUnitCore" class with a parameter ("Money Test"):

```
c:\> java org.junit.runner.JUnitCore MoneyTest
```

In this case, the results will also be displayed on the console, saying that the two tests were OK or not OK.

Fit ("framework for integrated test")

Another tool for unit testing is *Fit ("framework for integrated test")*. It was created by Ward Cunningham, the father of the WikiWikiWeb ("wiki"). The idea underlying Fit is to enhance communication and collaboration between developers and customers. "During development, how can customers know that their programmers are producing the right thing? How can programmers know what the customers really want? How can testers know what's right and what's wrong? Getting these groups to communicate effectively and precisely should be a goal for teams creating great software." [Cunningham 2005].

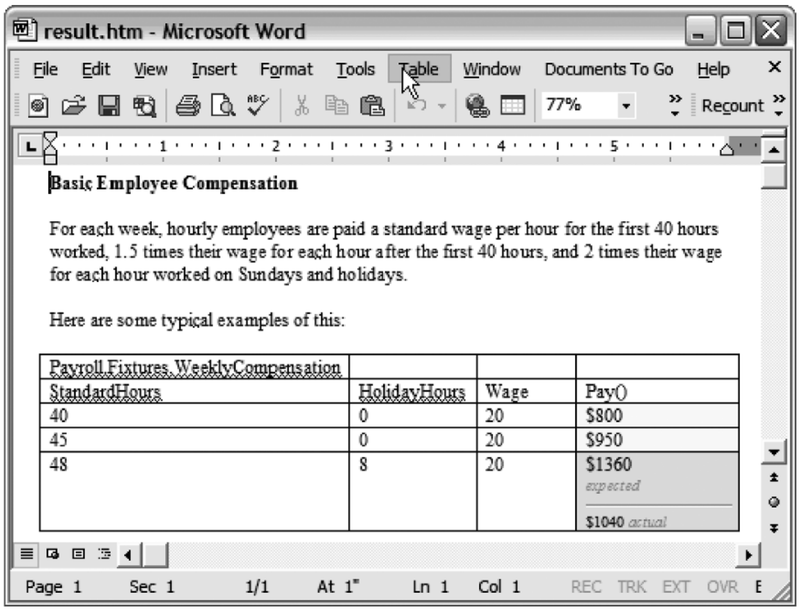
Test examples provided in MS Excel or Word tables

Fit strives to make testing visible and understandable not only for the programmer but also for the customer. The mechanism to achieve this goal is a web GUI for the test tool. Test cases are displayed in HTML tables, for example, and the results are also shown in the table. User involvement is supported by the fact that users can provide rule-style test examples in Excel or Word tables. These tables can be imported into the Fit tool. The programmer's job is then to define the structure of the tests in the underlying programming language, i.e. defining what the columns and/or rows of the tables stand for and how they are related.

In the example given in figure 6-26, the program under testing is to calculate employee pay. The customer created a Word table showing through examples how hourly pay should be calculated. The programmer imported this table into the Fit tool and wrote the tests to be run. The tests are executed when the HTML page is loaded, displaying the test results immediately.

The table in figure 6-26 shows the test examples and the results. The first row contains information for Fit on where the tests are defined. The second row has the headers for the columns. The remaining rows provide data examples. The first example says: "If someone works for 40 standard hours and zero holiday hours, and is paid $20 per hour, then his or her total pay is $800." [Cunningham 2005] Test results are displayed in colors. Green stands for OK, red for an error, yellow for an exception, and gray says that the cell was ignored for some reason. Obviously the expected result in the last row of the table is 1360, but the payroll program produced 1040 – an error.

Figure 6-26 **Fit testing example [Cunningham 2005]**


result.htm - Microsoft Word
File Edit View Insert Format Tools Table Window Documents To Go Help
77%
Recount
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR

Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

Mocking up the environment

Module testing has to cope with the problem that a module is usually just a small part of the overall software system. It interacts with other modules, e.g. passing messages to and receiving messages from these modules. How can the programmer responsible for the module examine the proper functioning of this single piece from the whole puzzle?

Mock object (mocks)

Unless module testing is implicitly embedded in integration testing – usually not a good idea but sometimes done this way – the answer is *mock objects (mocks)*. Mocks are substitutes for the immediate neighbors of the module to test – in an object-oriented language, classes containing test implementations that "simulate" the behavior of the actual classes. Mock objects have the same interface as the objects they represent, but not a complete implementation. In fact, they often have very simple implementations[§].

One problem resulting from this is that mock objects limit the degree of test coverage. Consider the case of a method invoked with an argument "product ID" and returning the stock of this product. A mock will probably always return the same stock value, leaving open all other cases (cf. above equivalence classes, boundary values, product ID does not exist etc.)

Automated tools for generating and running mocks

In a large system with many classes, writing mock objects for unit testing can be a time-consuming work. The more the current unit depends on other classes, the more mocks have to be written. To facilitate this work, automated tools for the generation of mock objects and running the mocks within unit tests have been developed. Many of them are open-source tools that can be downloaded from the Internet.

Integration testing

Testing and integration go hand in hand

While module testing looks at the individual pieces of the system under development, *(module) integration testing* brings the pieces together. The work to do in this stage is actually "integration" (of the modules). However, integration hardly ever works on the first attempt, requiring plenty of testing at the same time. Since testing and integration usually go hand in hand, "integration" and "integration testing" have more or less become synonyms.

Consider a layered architecture such as a three or four-tier architecture (cf. section 3.2.2) with 10 or 20 modules on each layer that have passed their unit tests. Several dozens of modules now have to be integrated, and the question is where to start integrating. Basically there are four options:

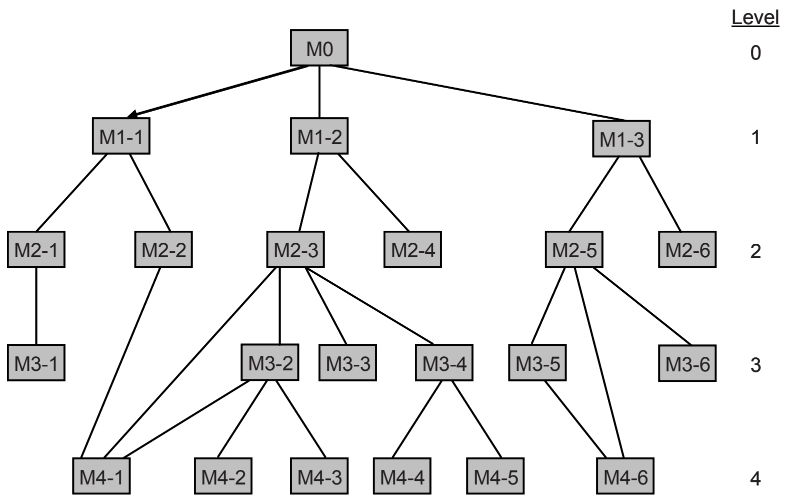
"Big bang" integration

1. *"Big bang":* Take *all* modules of the system, compile them, build an executable program and run it. What will happen in near to 100

§ For example, many mock implementations of methods expected to return a Boolean value contain just one statement ("return true.")

% of non-trivial cases is that the program crashes. The actual cause of the crash will be very difficult to find, since the initial mistake that caused the crash may have happened many modules away, in any of the other dozens of modules. Obviously a step-by-step integration approach is better in order to be able to localize errors.

Figure 6-27 **Dependency graph of a layered software system**



Step-by-step integration

Step-by-step integration can start from the top layer or the bottom layer (cf. figure 6-27). In a large system, not all modules of a layer will be integrated in one – even though smaller, yet still quite big – "bang" but in an incremental way. Since localizing errors in a large system can be a cumbersome task, due to the complex interactions in such a system, it is a good practice to start with a small subset of the modules and then integrate new modules in small steps, one by one. This makes it easier to spot the cause of an error.

For the stepwise integration, a call graph or better, a *dependency graph* (showing all dependencies between modules) can be used. Some modules at the leaves of the graph (or underneath the root, depending on the integration strategy) will be integrated first with their immediate neighbors, before proceeding bottom-up (or top-down, respectively). In the scheme of figure 6-27, the levels of the graph are numbered 0 to n.

Bottom-up integration

2. *Bottom-up integration:* Take a module of the second-lowest level (level n-1) of the dependency graph and integrate it with all modules of the lowest level that are invoked, or that the module depends on in some other way. In figure 6-27, for example, the first modules to integrate could be M3-2 with M4-1, M4-2 and M4-3. If the number of modules is too large, start with a few modules, use mocks for the other ones, and substitute the mocks incrementally by real implementations. Then take the next level n-1 module and do the same (e.g. M3-4 with M4-4 and M4-5), etc. Afterwards continue with a level n-2 module and integrate it with lower level modules (e.g. M2-3 with M4-1, M3-2, M3-3 and M3-4), etc. Once the top-level module has been successfully tested, integration testing ends.

Top-down integration

3. *Top-down integration:* Take the root module and integrate it with all modules that are invoked or that the root depends on directly. In figure 6-27, these are the modules M1-1, M1-2 and M1-3. If the number of level-1 modules is too large, start with a few modules, use mocks for the other ones, and substitute the mocks incrementally by real implementations.

Note that modules of level 1 (or lower) depend on other modules which have not yet had integration testing. Therefore the level-1 (and lower) modules must use mocks for level 2 (or lower) modules. In figure 6-27, M1-1 will need mocks for M2-1 and M2-2, M1-2 will need mocks for M2-3 and M2-4, and M1-3 will need mocks for M2-5 and M2-6.

On the next level, the process is the same. M1-1 will be integrated with the real M2-1 and M2-2, which in turn will need mocks for M3-1 and M4-1, respectively. Integration testing ends when the level n-1 modules have been successfully tested.

"Sandwich testing"

4. "*Sandwich testing":* Combine bottom-up and top-down integration testing. In many practical situations, testing is done in neither an exclusively top-down nor bottom-up manner. Combining top-down and bottom-up means starting with low-level modules, integrating bottom-up, and with high-level modules, proceeding top-down, at the same time or intermingled. For example, one tester can start from the top, the other one from the bottom, meeting in the middle. Top-down and bottom-up testing can also overlap, and sometimes even oscillate upwards and downwards.

Value-based integration testing

In addition to these formal perspectives, integration testing can be looked at from the customer's point of view, or from the perspective of which part of the system creates the most value for the organization. Integrating such modules that provide the most valuable functionality first can be a reasonable approach, if such functionality can be isolated in the dependency graph. In figure 6-27, let us assume that the branch beginning with M1-2 provides the most important functionality. Then the integration testing may focus on the branch including all modules from M1-2 downwards as far as M4-1 to M4-5, before less important functionality is tested.

System testing

Testing against the requirements specification

The last part of module integration testing is actually already a system test, i.e. a test of the entire system. However, the term "system testing" is normally used for testing the complete system against the *requirements specification.* This stage of testing is still the responsibility of the organization developing the information system, not the customer's responsibility.

System tests are black-box tests, investigating both functional and non-functional requirements. The "black box" in system testing is the complete software system, whereas in unit testing the black boxes are individual modules (units).

Functional requirements

For the testing against *functional requirements,* a good basis to develop test cases are the use cases and other UML diagrams from early development stages (e.g. sequence diagrams).

Non-functional requirements

Non-functional requirements are sometimes tested in separate tests with individual names. Examples include:

– *Performance testing:* Is the system's performance acceptable when it is run under a heavy load? Testing the performance requires that a representative workload is created for the tests – also called an *operational profile*. This is a set of test cases that reflect the actual mix of work that will be handled by the system [Sommerville 2007, p. 546].

– *Stress testing:* How will the system behave when it reaches, or exceeds, its regular operational capacity? Will it fail or degrade gracefully, i.e. continue to operate but at a lower speed? At which load level does the system fail and what happens when it crashes? To what extent is the system failure safe, preventing loss of data?

– *Recovery testing:* Is the system able to recover from program crashes, hardware failures, power failures and similar problems (e.g.

a network failure during data transmission)? How successful is the recovery, i.e. which state is the system able to restore?

– *Usability testing:* Can the system be adequately used regarding its intended purpose? Usability testing requires test cases that reflect realistic work situations.
– *Accessibility testing:* Does the system meet accessibility requirements, and to what extent (level "A", level "double-A", level "triple-A" conformance [W3C 1999])?

Before the system can be delivered to the customer, the developing organization has to make sure that all requirements are met and/or all contracted features are available and functioning. This may include integrating hardware and software if the system needs special hardware for operational usage.

Installation and interoperability testing

Testing on the target configuration

When the system testing is completed, the final system is delivered to the customer. Before it can go into operation – and before the customer will pay the bill – more testing is necessary. Normally the target hardware and software configuration under which the system will have to run is different from the configuration under which the system was developed. Therefore the system has to be ported to and installed on the target configuration. Testing the proper working on the target configuration is called *installation testing*.

Interoperability with other IS

Closely related with installation testing is *system integration testing (interoperability testing).* In today's digitally enabled firms, the delivered system is most likely just another piece in the organization's information systems landscape. It will need to work together with other information systems. For example, if the system under consideration adds agile manufacturing-planning capabilities to the organization's information systems, then it will probably have to interoperate with the ERP system, with technical information systems (such as CAD, CAP and CAM), with shop-floor control systems, with the CRM system and with other systems.

The purpose of system integration testing is to examine if the delivered system is compatible with existing systems, and how well it interoperates with them. This type of testing is usually performed by people from the customer's and the developers' organizations together.

Reengineering and integration of legacy software

System integration testing may require preparatory work on the other systems' side. These systems may not be ready to collaborate with the new system. Making them interoperable as well can be a costly process, in particular when the systems are *legacy systems*. Before integration can be considered, the legacy system may need to undergo a reengineering.

Since software reengineering is a significant effort of its own, corresponding projects should be started on time. Otherwise interoperability with the new information system cannot be tested. Reengineering and integration of legacy software are discussed section 7.5.

Acceptance testing

Testing by the customer

Finally the customer has to accept the system. In order to raise confidence that the new system will do its intended work properly to a satisfactory level, the customer's staff will perform their own tests, alone or together with personnel from the developers' organization.

Testing with real data

Most testing before was done with test data. Now the tests are performed with real data. Since acceptance testing is often the first time that the customer inspects the new system really thoroughly, it is not unusual that insufficient or inadequate requirements are detected. The closer the customer was involved in the development (as it is the case in XP, for example; cf. section 4.4.1), the fewer problems can be expected. In a traditional waterfall-like approach, problems such as the one illustrated in figure 4-4 are often observed.

Acceptance testing may also reveal that the performance of the system is unsatisfactory. Although the performance may have been tested during system testing, this was done with the help of test data. Now when real data are used, bottlenecks may show because the operational profile for performance testing might not have been adequate.

Beta testing

A special case of acceptance testing is the so-called *beta testing*. This term is used when new standard software or a new release of such software is being developed. The software company producing the system gives an almost final version of the system to selected customers. These customers agree to use the software and report back errors they detect [Sommerville 2007, p. 81]. The software producer gets valuable feedback from the user community and can remove errors that were not found during the internal testing stages.

But what does the customer get? The newest and utmost state-of-the-art product in the application field – something that software developers and "techies" seem to be fond of.

6.3.5 Debugging with a Lower CASE Tool

What is a "bug"?

Debugging is a common term in implementation and testing. "Bug" is a colloquial expression among software developers for any software error that prevents the program from behaving as intended, e.g. making it crash or producing an incorrect result. "De-" bugging literally means *removing* the bugs, but the way this term is commonly used includes *finding* the bugs as well.

Debugging occurs in all approaches to implementation and testing, in test-driven development as well as in conventional coding and testing. The more code has to be examined, the more difficult the debugging is. Debugging tends to be easier in TDD, because the additional code snippets to be tested incrementally are small. When a large piece of software has to be searched, a lot more effort is required. What makes debugging difficult is the fact that the cause of an error can lie anywhere in the software (the so-called "spaghetti bowl" syndrome). It rarely lies exactly in the code piece where the program failed or misbehaved.

The major steps in debugging are the following:

1. Find out that a bug exists
2. Locate the program statement where the bug occurs
3. Identify the cause of the bug
4. Fix the bug
5. Test the debugged program

Conventional debugging

In conventional coding and testing, programmers used to add debugging statements to the code displaying the values of critical variables and/or messages saying which modules, methods, branches etc. the flow of control went through. In the early times of computer programming – and sometimes still today, in very tough cases – the programmer would even print out a memory dump reflecting the state of the machine when the program crashed. (The reader can imagine that reading and interpreting a binary or hexadecimal coded memory dump is a challenging task!)

Today, the above mentioned debugging steps are supported by automated tools, called *debuggers*. Typical functions of a debugger include:

- Running the program statement by statement (stepwise execution),
- Stopping execution at defined points (so-called breakpoints) to examine the current state of the program,
- Tracking values of certain variables,
- Modifying the state of the program while it is running (e.g. setting new values).

Modern IDEs support debugging

A good debugger can significantly raise programmer productivity. Many powerful debuggers are available today. Modern lower CASE tools (IDEs) include effective debugging support. A typical IDE provides debugging information and features such as the following:

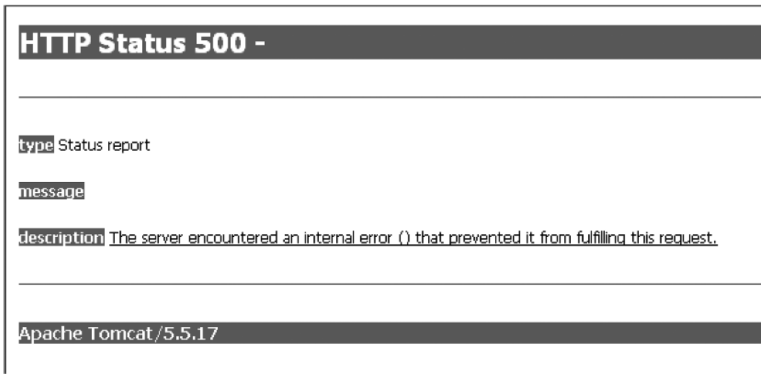
» *Call stack* – the sequence of method calls made during execution of the current thread. The last method invoked is usually shown on top.
» *Threads* – all threads of the running program (i.e. in the current debugging session).
» *Loaded classes* – the hierarchy of all classes currently loaded by the program, showing class names, methods, fields etc.
» *Local variables* – all local variables within the scope of the current method (variable name, data type, current value etc.).
» *Watches (data watches)* – specific variables and expressions defined by the programmer to watch while debugging (e.g. variable or expression name, data type and current value).
» *Breakpoints* – points in the program where the execution is paused for debugging purposes. For example, a breakpoint is often set before the statement where the program failed, in order to let the programmer watch its behavior from that point on.
» *Console* – messages from the debugger about the current session (e.g. program execution state, debugging state).

Debugging example

Debugging is a multi-step process that requires experience and in-depth programming knowledge. Within the limited space of a printed book, we can only give a short outline of what debugging activities are like. For illustration, we use a very simple example. However, the reader should keep in mind that debugging in practice is a time-consuming task.

Consider the following situation: In the implementation of the *advertisement-ordering system* used as an example before, some developers were responsible for the core Java classes (cf. figure 5-53) and others for the graphical user interface (cf. figure 6-10). Since both teams were working in parallel, the GUI developers needed mocks of the underlying system classes (e.g. "Edition", "Customer", "Advertisement" etc.). These mocks are quite simple, mostly returning just null or constant values.

Figure 6-28 **A useless error message**



HTTP Status 500 -

type Status report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

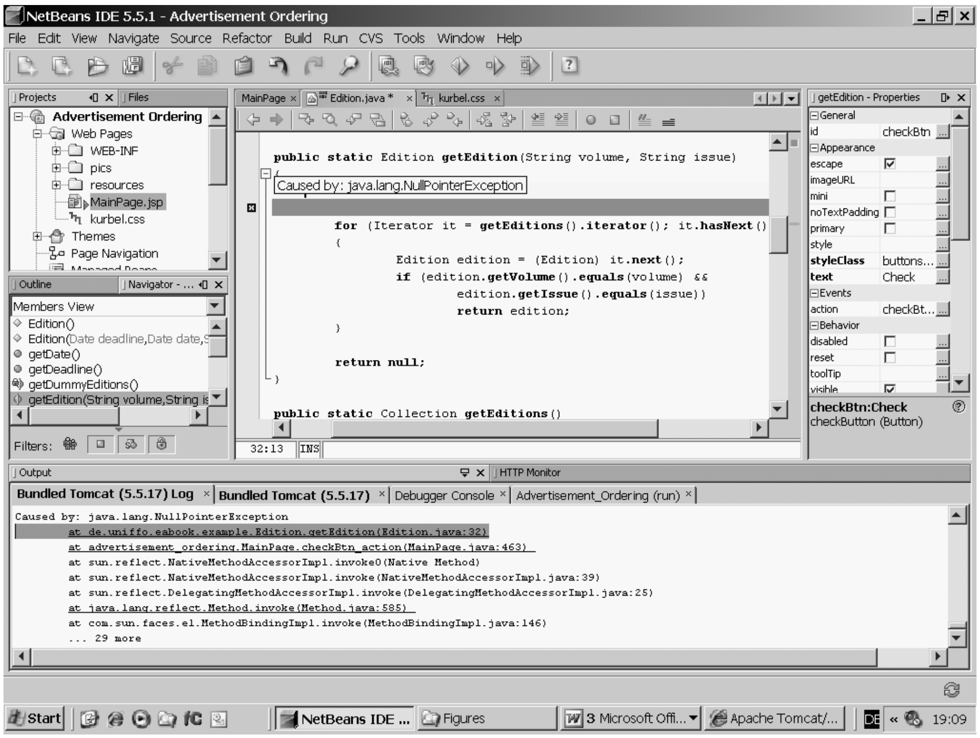
Apache Tomcat/5.5.17

The following problem occurred: When the GUI was tested, the web browser displayed a window with a rather useless error message shown in figure 6-28. In order to find out the cause of the error, the developer ran the program in NetBeans. The screenshot in figure 6-29 reflects a typical work situation during testing. The output window at the bottom contains log messages from the Java runtime system and the web server. The main window in the middle shows Java source code that is being executed.
The output window contains a very long list of log statements from which only the last text block is visible. This text block describes the final error situation as follows:

```
Caused by: java.lang.NullPointerException
   at de.uniffo.eabook.example.Edition.getEdition
   (Edition.java:32)
   at advertisement_ordering.MainPage.checkBtn_action
   (MainPage.java:463)
   at sun.reflect.NativeMethodAccessorImpl.invoke0
   (Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke
   (NativeMethodAccessorImpl.java:39)
```

```
at sun.reflect.DelegatingMethodAccessorImpl.invoke
(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:585)
at com.sun.faces.el.MethodBindingImpl.invoke
(MethodBindingImpl.java:146)
... 29 more
```

Figure 6-29 **Program execution with error (NetBeans GUI)**



```
public static Edition getEdition(String volume, String issue)
{
Caused by: java.lang.NullPointerException

        for (Iterator it = getEditions().iterator(); it.hasNext()
        {
                Edition edition = (Edition) it.next();
                if (edition.getVolume().equals(volume) &&
                        edition.getIssue().equals(issue))
                        return edition;
        }

        return null;
}

public static Collection getEditions()
```

```
Caused by: java.lang.NullPointerException
        at de.uniffo.eabook.example.Edition.getEdition(Edition.java:32)
        at advertisement_ordering.MainPage.checkBtn_action(MainPage.java:463)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:585)
        at com.sun.faces.el.MethodBindingImpl.invoke(MethodBindingImpl.java:146)
        ... 29 more
```

The interesting lines of this text block are the first three ones. A null-pointer exception occurred (something that usually happens when an object was not initialized) when the "getEdition" method of the "Edition" class was executed (in statement 32 of this class). "getEdition" was called by the "checkBtn_action" method of "MainPage" (in statement 463 of "MainPage.")

A null-pointer exception

Clicking on the top "at" line in the output window makes the debugger go to the statement where the program failure occurred and position the code editor to this statement. This is the highlighted line in the middle window. The program crashed when trying to execute the "for" statement.

Actual debugging begins here

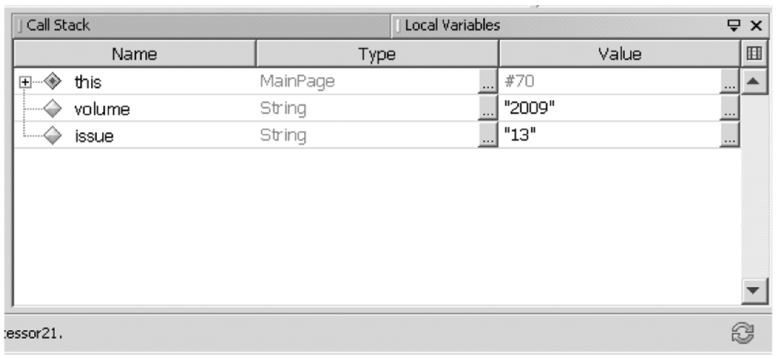
Why? Nothing seems to be wrong with the "for" statement. Like in most cases, the cause of the problem is not the statement where the failure became visible but somewhere else. How to find the erroneous statement? This is the crucial question, and this is actually where the *debugging begins.*

To localize the origin of the problem, the developer starts the program in the debugging mode and executes it statement by statement. A number of NetBeans tools supporting this process are available. In particular, breakpoints allowing stepwise execution, a trace of the invocations (call stack), and the values of relevant variables are helpful in a debugging situation.

Local variables

To solve the error-location problem, a number of breakpoints were set, and some variables and the call stack were watched in the output window. Figure 6-30 shows the values of *local variables* in the enlarged bottom right corner of the debugger screen. The values of "volume" and "issue" ("2009" and "13") are the values the tester had entered in the respective text fields of the web form.

Figure 6-30 **Local-variables window in NetBeans debugger**



Name	Type	Value
this	MainPage	#70
volume	String	"2009"
issue	String	"13"

Call stack

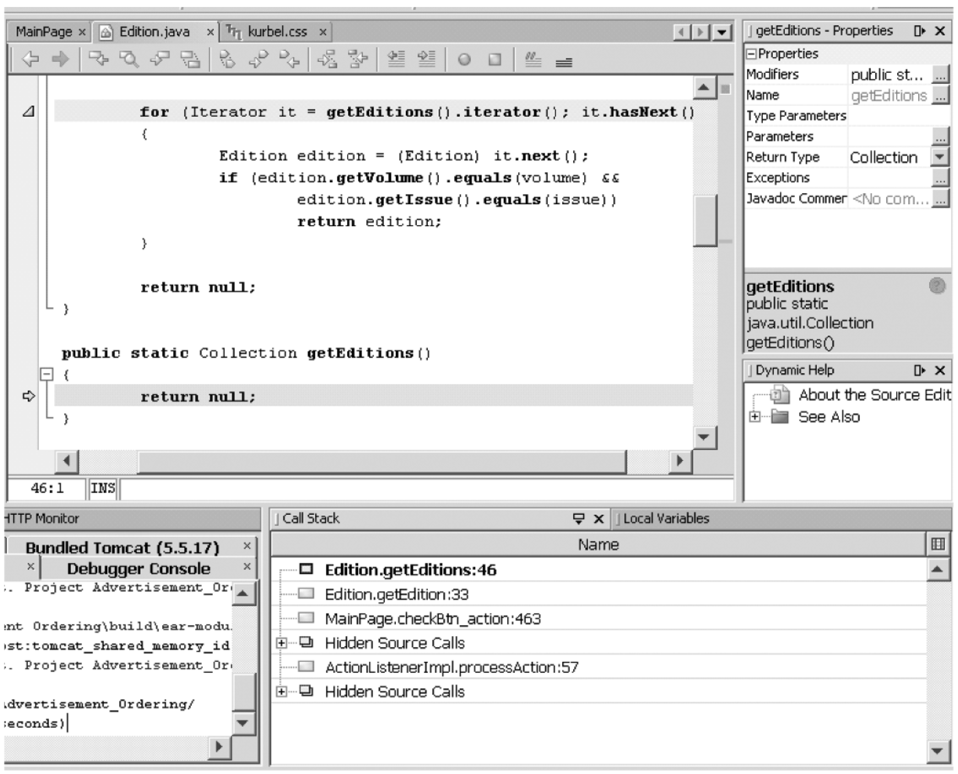
The call stack displayed in the call-stack window and the code section corresponding to the last entered method are shown in figure 6-31. The current statement to be executed is found in the "getEditions" method, in line 46 of the "Edition" class ("return null"). This method was invoked from the "getEdition" method in line 33. The "getEdition" method itself was called in MainPage at line 463, i.e. from the method "checkBtn_action".

Two more steps past the snapshot situation of figure 6-31, the program terminates with a null-pointer exception. The first step is execution of the "return" statement (i.e. returning the "null" value to the "getEditions" invocation in the aforementioned "for" statement). The second step is execution of the "for" statement (and indirectly of the embedded "getEditions" call).

Why did the program crash?

Now, why did the program crash? The answer is that the GUI developer forgot to return an actual value from a "getEditions" invocation when he wrote the "Edition" class mock. Methods calling "getEditions" expect to receive an object in return, but they receive a null value instead. This is why the null-pointer exception was raised.

Figure 6-31 **Call-stack and code windows in NetBeans debugger**



```
        for (Iterator it = getEditions().iterator(); it.hasNext()
        {
                Edition edition = (Edition) it.next();
                if (edition.getVolume().equals(volume) &&
                        edition.getIssue().equals(issue))
                    return edition;
        }

        return null;
    }

    public static Collection getEditions()
    {
        return null;
    }
```

Properties	
Modifiers	public st...
Name	getEditions
Type Parameters	
Parameters	
Return Type	Collection
Exceptions	
Javadoc Commer	<No com...

getEditions
public static
java.util.Collection
getEditions()

```
. Project Advertisement_Or

nt Ordering\build\ear-modu
st:tomcat_shared_memory_id
. Project Advertisement_Or

dvertisement_Ordering/
econds)
```

- **Edition.getEditions:46**
- Edition.getEdition:33
- MainPage.checkBtn_action:463
- Hidden Source Calls
- ActionListenerImpl.processAction:57
- Hidden Source Calls

Once the source of the problem was found, fixing was simple. Instead of returning "null", a dummy object was returned[§]. From then on the "getEditions" method was able to continue its work without interruptions (at least during the testing stage). Correct execution (i.e. a correct result of the program) was shown earlier in figure 6-10.

§ The statement "return null" was substituted by "return getDummyEditions()", a mock method that returns an "Edition" object with constant values.

7 Implementing Standard Software

As we pointed out at the beginning of the chapter 6, the term "implementation" has several meanings. Up to and throughout chapter 6, implementation was used in the computer-science oriented sense, meaning the realization of a design through "lower-level" techniques such as programming and creating database schemata.

"Implementation" as organizational term

In organizational theory, implementation stands for introducing a new concept or solution into the organization and bringing it to life. In naming this chapter "implementing standard software", we are referring to implementation as an organizational term: putting standard software into operation, including all necessary activities to achieve this goal.

7.1 Standard vs. Individual Software

Build, buy, or rent?

An early decision to be made in an information system development project is whether to "build, buy, or rent". This question was addressed in section 2.2.3. While chapters 4 to 6 dealt with the "build" case (i.e. the system is developed by the organization), the "buy" case will be discussed in this chapter.

The number of organizations that develop comprehensive information systems themselves is decreasing. In typical business settings, large-scale information systems are mostly systems that were developed by specialized software companies as standard software and not by the user organizations.

Individual ISD is too expensive

Some organizations still create individual information systems although standard software is available. However, these are primarily large companies that have their own software development departments. For most other organizations, individual software development has just become too expensive. Taking into account the high development cost and the need for well-trained personnel with up-to-date technological know-how, the majority of user organizations prefer to acquire ready-made software.

The increasing market share of standard software is also due to the fact that nowadays this software can be better adapted to an organization's individual needs than used to be the case in the past.

Loss of competitive advantage due to standard software?

In the beginning, standard software was primarily used for non-core business functions and processes that lent themselves easily to standardization such as financial accounting, payroll and bill-of-materials processing. However, organizations were hesitant to base their core processes on standard software because they were afraid of the standardizing effect of this software. This is because companies often differentiate themselves from their competitors through the implementation of core processes. Standard software was seen as an "equalizer", bearing the risk that competitive advantages embedded in the core processes would be lost.

Nowadays, organizations increasingly employ standard software for their core business processes as well. One reason for this is that the technological means to individualize standard software have become quite powerful. Therefore, organizations can use standard software, adapt it to their particular needs, and still differentiate themselves from their competitors. Advantages and disadvantages of standard software are summarized in figure 7-1.

Individualized standard software

Figure 7-1 **Advantages and disadvantages of standard software**

Advantages	Disadvantages
Cost Predictability Timeframe Availability Best practices, software quality Maintenance, support and further development	Customization effort Integration effort Vendor dependency Oversize system Organizational changes

Advantages of standard software

➠ *Cost:* The main advantage of standard software, but by far not the only one, is *cost*. While the cost for an individual software-development project has to be completely born by one organization, the cost for developing standard software is passed on to many organizations, each one paying a price to the vendor which is only a fraction of the total development cost.

Development cost is passed on to many organizations

➠ *Predictability:* The cost of standard software is not only lower but also more predictable than the cost of individual software. While the price of standard software is known in advance, cost estimation for individual software development is subject to significant uncertainty (cf. section 2.4.3). Likewise, the cost of implementing the standard software in the organization can be estimated based on the vendor's previous experience with similar projects. Comparable estimates are not available for an individually developed software system.

Package price is known in advance

➠ *Timeframe:* Practical experience has shown that many ISD projects exceed their deadline, because it is very difficult to predict the duration of software-development activities. Implementing a standard software system is easier to oversee. This is because the time needed to develop the software is not required. Only the duration of the customizing and implementation parts, based on the vendor's experience, need to be estimated.

➠ *Availability:* A standard package is essentially available immediately, whereas an individual software system will be available months or years from now. For competitiveness, quick availability is often an essential requirement.

Continuous improvement

➠ *Best practices:* A standard-software vendor receives customer feedback regarding drawbacks, missing features and things to improve. Such feedback is usually incorporated into future system releases, enhancing the system gradually. The more customers the vendor has, the more know-how will eventually be included. Business information systems such as ERP systems incorporate industry-wide knowledge and best practices which an individual organization otherwise would not have access to.

➠ *Software quality:* In the same way, the software-technical quality of the system improves over time as many users point out errors, performance flaws and shortcomings regarding non-functional software requirements that the vendor will remove.

➠ *Maintenance, support and further development:* Not only is the user organization relieved from these time-consuming activities but they also do not need to worry about keeping up with the latest technologies. These are tasks the vendor will solve.

Disadvantages

Customization is expensive

➠ *Customization effort:* Since it hardly ever happens that standard business software meets exactly the organization's needs, the software almost always has to be adapted or extended (cf. section 7.3). This requires significant effort, takes time and costs a lot of money. Many organizations "... make the mistake of over-customizing their application modules in attempt to appease the end-users of the systems. This tendency to over-customize can be expensive and consume internal resources." [Beatty 2006, p. 108] A good deal of the cost shows later when customizations must be carried from one version of the system to the next upgrade.

Legacy and other systems

➠ *Integration effort:* Connecting the standard software with existing information systems, possibly legacy systems, in the organization's IS landscape requires significant effort in order to bridge the gaps (cf. section 7.4).

Switching cost is prohibitive

➠ *Vendor dependency:* The user organization depends on the vendor regarding all aspects of the information system, including availability of new features and releases, the risk of vendor insolvency and/or acquisition by a competitor, the vendor's product, marketing and technology strategies etc. Once a comprehensive standard software system has been implemented, the cost of switching to a different vendor is usually prohibitive.

Standard software contains more than the organization needs

➡ *Oversize:* Standard software is often oversized because it contains more functionality than an individual organization needs. In order to be able to serve the requirements of a variety of customers, the vendor usually includes a wide array of functions from which the individual customer can select the most suitable (cf. section 7.3.1). Nevertheless, the system contains more than the customer needs, which often results in increased hardware requirements.

Adapting the organization to the software

➡ *Organizational changes:* Implementing standard software – with standard business processes and rules – in an organization usually means that the organization, to some extent, has to adjust to what the software prescribes. This is not what most organizations want. They would rather have the software reflect the exact processes and rules the organization ran before. Although listed under "disadvantages" here, the need to adapt also has positive effects, as standard software often introduces organizational improvements (see "best practices" above).

7.2 Process Models for Standard Software Implementation

Implementing standard software is a costly process with time-consuming activities. Some activities are the same as in developing an information system, but most differ in that the complete information system already exists.

Usually a number of similar systems are available on the market. Being quite large and containing thousands of functions, a major challenge for the buyer is to understand these systems and how well they would satisfy the organization's needs.

Some process models are supported by tools

Since the implementation of standard software requires a lot of effort and has a long-term impact on the organization, process models were developed to support the implementation process. Comprising a large number of activities to be performed in a specific order, some process models are supported by tools facilitating the implementation work. In section 7.2.3, an example of a vendor-specific process model with tool support will be given for illustration.

7.2.1 Selecting the Vendor and the Product

The first stage, or one of the first stages, of a process model is the selection of an appropriate standard software system for the problem domain under consideration. Usually several options are available. In some areas, the number of systems to select from is quite large. For example, several hundred standard packages for enterprise resource planning are on the market. Although some address specific industries or company sizes, the number of candidates a typical business organization can select from still amounts to several dozen.

Checklists

Checklists can be very long

Just as in information systems development, the starting point for the selection process is a requirements specification. In order to make heterogeneous software offerings comparable, many user organizations develop *checklists* containing the business processes and/or functions they want supported by the software. Such checklists can range from several dozen to thousands of items. For example, Trovarit AG, a consulting firm specialized in the implementation of enterprise resource planning, uses a checklist of 2,250 criteria for tool-supported selection among 600 ERP and other business systems (http://www.trovarit.com/).

Smaller checklists tend to address the desired functionality in an aggregated form, whereas long checklists usually exhibit much detail. Since an aggregated description of a specific functionality is often not sufficient, stakeholders tend to elaborate the checklists in depth.

For example, a checklist for a *dispatching system* might specify that vehicle routing functionality is needed. Such a general requirement would probably be met by almost every dispatching system on the market. However, if the organization says that they want an optimization algorithm for the routing and automated processing of RFID data for the monitoring of actual shipments, the list becomes longer (and the number of candidate systems smaller).

Figure 7-2 Excerpt of a checklist for APS software [Homer 2007]

Ref	Software Product Functionality	Field Type	*Supplier1 Product 1*	*Supplier 2 Product 2*	*Supplier 3 Product 3*
	⋮				
49	Manufacturing Planning & Scheduling:				
50	Regenerative Schedule	Y/N	Y	Y	Y
51	Incremental Schedule	Y/N	Y	Y	Y
52	Resources/Constraints that can be modeled:				
53	Labour	Y/N	Y	Y	Y
54	Machines	Y/N	Y	Y	Y
55	Tools	Y/N	Y	Y	Y
56	Sub contractors	Y/N	Y	Y	Y
57	Materials	Y/N	Y	Y	Y
58	Shelf life of product	Y/N	N	Y	Y
59	Warehouse capacity	Y/N	N	Y	Y
60	Transportation	Y/N	N	Y	Y
61	Work Centres-machine/labour combination	Y/N	Y	Y	Y
62	Multiple plant sourcing	Y/N	Y	Y	Y
63	All of the above, simultaneously	Y/N	N	Y	Y
64	Modeling capabilities:				
65	Set up time	Y/N	Y	Y	Y
66	Run time	Y/N	Y	Y	Y
67	Wait time	Y/N	Y	Y	Y
68	Move time	Y/N	Y	Y	Y
69	Multiple time fences	Y/N	Y	Y	Y
70	Substitute resources/materials	Y/N	Y	Y	Y
71	Alternate routings i.e. machines	Y/N	Y	Y	Y
72	Rate based modeling	Y/N	Y	Y	Y
73	Fixed-duration modeling	Y/N	Y	Y	Y
74	Infinite Capacity Planning	Y/N	Y	Y	Y
75	Finite Capacity Planning	Y/N	Y	Y	Y
76	Floating Bottlenecks	Y/N	Y	Y	Y
77	By-products	Y/N	Y	Y	Y
78	Co-products	Y/N	Y	Y	Y
79	Variable production by part by machine	Y/N	Y	Y	Y
80	Operation overlapping	Y/N	Y	Y	Y
81	Split operations	Y/N	Y	Y	Y
82	Assigns tooling to operation	Y/N	Y	Y	Y
83	Schedule constrained by tooling availability	Y/N	Y	Y	Y
84	Variable delay to force op to start at start of shift	Y/N	N	Y	Y
85	Supports synchronisation of operations	Y/N	Y	Y	Y
86	Maintains high utilisation of bottlenecks	Y/N	Y	Y	Y
87	Supports sequence dependent scheduling of set ups	Y/N	Y	Y	Y
88	Supports scheduling of development jobs	Y/N	Y	Y	Y
89	Supports scheduling of maintenance jobs	Y/N	Y	Y	Y
90	Rules based approach for sequencing	Y/N	Y	Y	Y
91	Distribution & Inventory Planning				
92	Supply Network Definition:				
93	Supplier	Y/N	Y	Y	N
94	Plant	Y/N	Y	Y	N
95	Distribution Centre	Y/N	Y	Y	N
96	Customer Location	Y/N	N	Y	N
97	Supply network planning tools:	Y/N	N	Y	N
98	Linear programming	Y/N	N	Y	N
99	Heuristics	Y/N	N	Y	N
100	Multi plant sourcing logic	Y/N	N	Y	N
101	Optimise truck loads	Y/N	N	Y	N
102	Prodn sourcing, inventory build, transport balancing	Y/N	N	Y	N
103	Global supply chain design.	Y/N	Y	Y	N
104	Rules based order fulfilment	Y/N	Y	Y	Y
105	First come/first served	Y/N	Y	N	Y
106	Fair share deployment	Y/N	Y	Y	Y
107	Prioritised allocation	Y/N	Y	Y	Y
108	Forecast consumption rules	Y/N	N	Y	N
	⋮				

An example of a checklist for the selection of *advanced planning and scheduling (APS)* software for supply chain management is given in figure 7-2. It shows an excerpt referring to the manufacturing planning and scheduling part of APS. Entries for three candidate software systems were already made in this checklist.

Request for proposals (RFP)

When the organization prepares a request for proposals (RFP) to be given to potential vendors, more questions than those referring to the software system's functionality are added. Thus the RFP will typically consist of a long list of questions referring to topics such as:

– System functionality details
– Hardware and software requirements (incl. non-functional requirements such as scalability)
– Organization of service and support, service levels
– User training and help (hotline, help desk)
– Cost (license, upgrade, maintenance, training etc.)
– Legal issues (contract, indemnification, liability etc.)

Not all checklist criteria are of the same importance. Some definitely need to be satisfied ("must have"), while others would be "nice to have", but are not indispensable. "Must have" criteria can be applied in the selection process to eliminate candidates, unless the missing functionality can be obtained otherwise (e.g. by additional programming).

Disadvantages of checklists

Long and detailed checklists have been criticized for obvious disadvantages. Major drawbacks of the checklist approach include the following:

1. The more elaborate the desired functionality details are, the higher the risk that the potential for organizational improvement is missed. Stakeholders tend to coin the current way of problem solving into the checklist and thus prescribe it for the future solution. This means that organizational shortcomings are also transferred. Often the individual features have to be developed through additional programming, causing additional costs, and best practices embedded in the standard software are not adopted.

Many functions of a detailed checklist are not used later

2. Stakeholders tend to over-specify the desired system with functionality that appears desirable to them but is not really necessary. This means that even good systems that do not provide this specific functionality are either eliminated from the list of candidates, or if they survive on the list and one such candidate wins at the end, the special functions will be implemented at additional cost.

What is worse, many detailed requirements that dominated the selection process are later found to be not so important after all. Practical experience shows that the majority of functions specified in a detailed checklist are not used when the new system is in real operation.

3. The effort to create a detailed checklist is very high. It involves stakeholders from different departments, requiring long discussions, compromises and a balance of interests among them.

Spending too much time on creating checklists?

Taking these points into considerations, some consultants recommend not spending too much time on creating checklists for vendor selection if a sound basis of standard software is available on the market. In the ERP field, for example, surveys have shown that organizations can expect 90 % of their requirements to be satisfied by the top ten ERP systems, and even 95 % by the top three systems [Scherer 2004].

Utility-value analysis

The reader may imagine the difficulties of a checklist-based selection process by considering figure 7-2. If there were 50 APS systems on the market, then a checklist could help to reduce the number of candidates to a manageable size of perhaps three to five systems. Yet the remaining systems would still have their "Y" and "N" entries at different places. In the example, products 1 and 3 apparently do not provide supply network planning tools. Product 2 does include these tools, but suppose it costs twice as much as the other two products. Should products 1 and 3 be eliminated as candidates? Perhaps for other criteria they are superior to product 2, exhibiting "Y" entries where product 2 has "N"s.

Benefits and drawbacks of available/missing features

To come to a final decision, the benefits of having certain features must be weighed against the drawbacks of missing other features. Since most benefits and drawbacks cannot be measured directly in money or any other quantitative units, it is necessary to apply qualitative judgement. Methods supporting this approach often use utility values for the criteria provided by the decision maker.

Weighting criteria

In a typical *utility-value analysis,* all criteria are weighted with percentages reflecting their relative importance to the decision maker. For each alternative (in our case: for each candidate software system), each criterion is then assessed in such a way that points are given reflecting how well the candidate satisfies the criterion. Points can range, for example, on a cardinal scale from 1 (very bad) to 10 (very good). Multiplying the points by the percentages and adding up the products yields an aggregated utility value for each alternative. The candidate with the highest utility is the system to choose.

Assessing three standard packages

Figure 7-3 shows an example of the assessment step. System 2 was found to cover most of the desired functionality (90 %) while the other two systems cover only 70 % and 60 %, respectively. Therefore the customization effort is expected to be normal (4 person months) for system 2 but higher for the other two systems (10 and 12 person months, respectively). System 2 is widely established on the market and has over 500 installations, a matter from which a certain level of trust in the system may be derived.

Figure 7-3 Aggregated software assessment (example)

Criterion	**Product assessment**		
	Product 1	Product 2	Product 3
System functionality	70%	90%	60%
Non-functional requirements	very good	OK	quite good
Cost (license, hw/sw, maintenance)	1,200,000	1,750,000	1,150,000
Customization effort	10 pm	4 pm	12 pm
Technical service & support	excellent	average	mediocre
User training & help	average	good	good
Reference installations	26	> 500	80

pm = person months

Figure 7-4 Utility-value analysis

Criterion	**Weight** (%)	**Points from product assessment**		
		Product 1	Product 2	Product 3
System functionality	30	7	9	6
Non-functional requirements	10	9	6	8
Cost (license, hw/sw, maintenance)	20	5	2	5
Customization effort	20	4	7	3
Technical service & support	10	10	5	4
User training & help	5	5	7	7
Reference installations	5	2	10	5
Total	100	615	645	520

Mapping the assessment to utility values

The mapping of these qualitative and quantitative results to utility values (points from 1 to 10) is shown in figure 7-4. For example, the highest number of reference installations (> 500) was given 10 points while the 26 and 80 reference installations received only 2 and 5 points, respectively. Summing up the points weighted with their percentages yields total utility values of 615, 645 and 520. The winner is obviously product 2. If the organization bases its decision on the utility-value analysis, then it will license this system.

7.2.2 A Generic Process Model

Implementing standard software requires various activities before and after the actual implementation step, resulting in a multi-stage process. Some stages are the same as or similar to the stages of information systems development while others are quite different.

In this section, we discuss a typical process model that user organizations follow when they have decided to implement standard software. The initial perspective is the same as in chapter 4: the starting point for an implementation is an approved project proposal (cf. section 2.2.1). This means that the management decisions to launch the project and to license standard software instead of developing a system inhouse have been made.

We also assume that the organization is willing to do, or has already done, business process engineering and/or reengineering, i.e examining the current processes and elaborating potentials for improvement. Determining the new processes in detail is done later – when the final decision for the product to buy has been made.

Business IS are built under assumptions

The reason is that business information systems are usually built under assumptions including: how an organization works; what organizational structures and typical business processes look like; common data structures and database entities; workflows; etc.

Standard software changes the organization

Although the software vendor may have set the assumptions based on practical observation of a large number of applications, the individual organization probably has not worked exactly according to these assumptions up to now. For the organization it makes sense to adopt some (or all) of the best practices built into the standard software, instead of ignoring the underlying assumptions and changing the standard software so that it exactly maps the current organizational and process structures. (This would mean making individual software out of standard software.) However, adopting best practices and other features of the standard software usually requires organizational change.

The fine-tuning of business processes, functions and organizational structures depends on the particular software product chosen and on the particular assumptions the vendor of this product made. Therefore the details of the necessary changes can only be determined when the concrete product has been selected. We assume that this change process is going on in parallel to the software implementation process.

Typical process models represent basically a sequence of stages as shown in figure 7-5. Major tasks and stages are the following:

(1) Vendor and product selection – evaluating vendors and standard packages available on the market as discussed in section 7.2.1.

Choosing the modules to be implemented

(2) Configuration – choosing those modules of the selected product that should be implemented in the organization. The reason why standard software is also called "package software" or "modular program" is that it is usually composed of many modules combined in a package. Not every organization needs every module. A bank, for example, will not need the production-planning module of the chosen ERP system but rather the financial modules.

In some cases it is difficult to decide which modules to adopt and which not. Most functionality of the ERP system's materials management (MM) module will not be of any use for the bank because the primary material it manages is money. However, some functions of the MM module may still be useful, e.g. inventory management for office supplies. The question is whether it is worth licensing (and paying for) the MM module for this purpose, or is it better to use simple off-the-shelf inventory management software in addition to the ERP system. A trade-off between higher cost (licensing) and integration problems (using additional stand-alone software) has to be made.

(3) Negotiations and contract – settling the deal with the selected vendor. The price, terms and conditions are likely to be negotiated in parallel to the selection and configuration process. The vendor and the buyer will enter into the final agreement once the module configuration is chosen and all questions regarding support, services, training, maintenance, upgrades etc. are solved.

Infrastructure and installation

(4) Installation – setting up a working hardware and software configuration at the customer's site. Major activities include preparing the computer system on which the system will run, and installing the database

management system, the application server and the web server the standard software will use. If these infrastructure components are already available, they must be prepared to host the new system. When the infrastructure is working, the program files and all other software components making up the new system can be installed.

Testing if and how well the standard software works together with the rest of the organization's information systems landscape (*interoperability testing*, cf. section 6.3.4), can either be part of the installation stage or the customizing and extensions stage, depending on how much individual modification and extension work is required. If significant parts of the system are re- or newly programmed, then it is obviously not appropriate to test its interoperability before this code exists.

Customization

(5) Customizing and extensions – adapting the standard software to the organization's individual requirements. In most implementation projects, this is the stage that requires most of the effort. Customization and related topics will be discussed in detail in section 7.3.

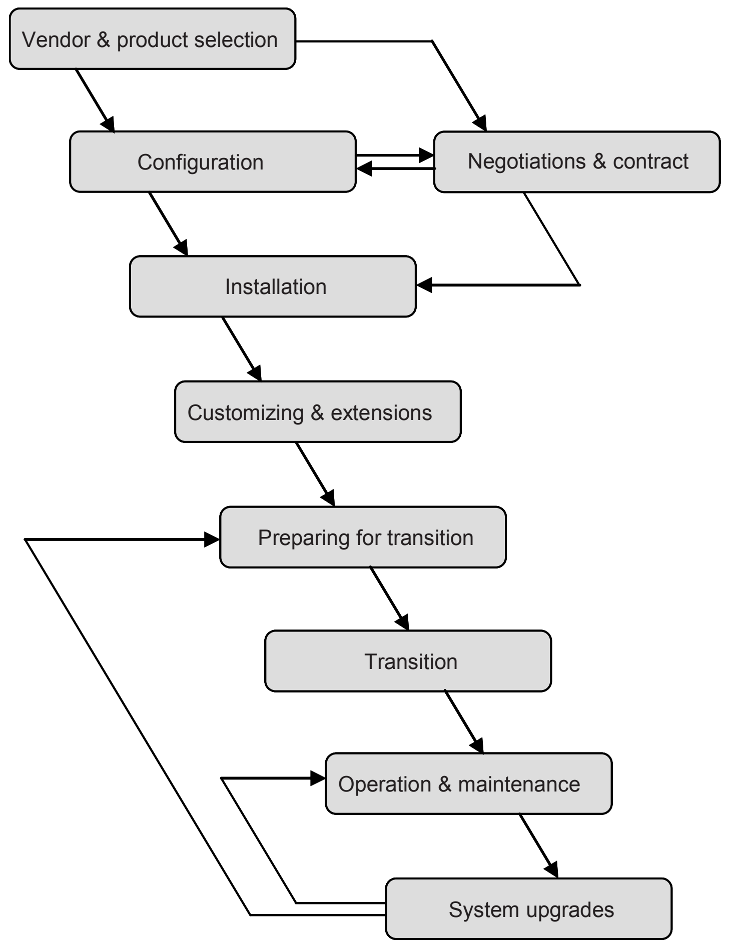
Making the organization fit for transition

(6) Preparing for transition – making the organization fit for the new system. This stage includes training of end-users and technical staff, producing organizational instructions and customized user manuals, and final testing, in particular acceptance and performance testing (cf. section 6.3.4). Before the system can go into operation, the stakeholders examine it against the requirements formalized in the requirements specification and/or the contract (acceptance testing). Since most work before was done using test data, the system's performance under "real" conditions – using real data and including extreme workloads – has to be examined to ensure that the system will not fail in daily operation.

It should be noted that the focus of the "preparing for transition" stage is not a general reorganization and reengineering of the business. The major tasks in preparing the organization for the new system, such as developing new organizational structures and establishing new business processes, have to start much earlier. They will be going on in parallel to the process stages described in this software implementation process model.

An important question to be answered is *how* the transition from the old system, or from the old way of doing things, to the new system will be organized. This step bears significant risk. Whereas the old processes and functions were known to work, the new, standard-software supported processes and functions have not proved their reliability yet. Therefore, the new solution bears a higher risk of business operations being disturbed or interrupted if the software fails to work properly. One of the three options described next has to be chosen.

Figure 7-5 Process model for standard-software implementation



(7) Transition – the actual implementation step putting the new system into operation. This may include moving from a pre-production hardware and software environment to the live system. In practice, the transition is often done according to one of the following options:

a) *Parallel operation* – implementing the new system parallel to the old system and running both systems for a while. All transactions are basically performed in the new system but redundantly in the old system as well. An obvious advantage is safety: If the new system fails, work can continue with the old system. The major disadvantage is also obvious: double work.

Running both the old and the new system for a while

b) *Stepwise transition* – starting the transition process by implementing only one or a few modules of the new system and implementing additional modules later, step by step. For example, implementation of an ERP system might start with the human-resources (HR) module and continue later with financial and managerial accounting, followed by materials management. An advantage of stepwise transition is that the organization can gain experience with the new system. End-users in the HR department learn and become acquainted with the system. Perhaps they find errors or drawbacks that the vendor can remove. If experiences are positive, the next module can be put into operation. If experiences are extremely negative, the entire standard-software deal might even be cancelled.

Implementing only a few modules at first

c) *"Big bang"* – defining a cut-over date and replacing operations with the old system by the new one. At the specified date, all users migrate to the new system, and operation of the old one is discontinued. This type of transition is extremely risky because many things not thought of before can go wrong. However, if it works, the organization can benefit from the new solution immediately.

Replacing all at once

The majority of practical implementation projects follow the second approach – stepwise transition. However, a consequence of proceeding in steps is that the implementation of a large standard-software system can take years.

The big-bang approach, even though it is very risky, can work if it is prepared very carefully. There are organizations that have successfully employed it. Preconditions are extensive user training and thorough acceptance and performance testing. Powerful help features, a hotline and extensive technical support available from the cut-over date on can help to reduce and/or remove unforeseen problems quickly.

(8) Operation and maintenance – using the system for its intended purpose every day. In this stage, it is common that flaws and errors are observed. The organization responsible for removing them is either the software vendor, if the error occurs in a standard module, or the user organization themselves, if the error occurs in code that was changed or created by their IT staff (such as an extension module). The more

changes to and extensions of the vendor's code made and the tighter the module coupling, the more likely it is that the parties will argue about whose fault it is that the error or flaw occurred.

New releases

(9) System upgrades – adopting new versions of the standard software. Most software vendors work in cycles. For some period of time, they collect new requirements, shortcomings observed by the users and fundamental errors that cannot be removed as part of maintenance. Then they incorporate the corresponding changes into their system. They also add new features in order to become more competitive. A new, upgraded version of the system (usually called a *release*) is produced and offered to the customers. New hardware or software technologies, such as a new software architecture, operating system or GUI technology may also trigger the development of a new version.

The customer can choose whether to adopt the upgrade or not. Software vendors often put pressure on their customers to implement the new release. However, many customers are hesitant to do so. Unless they are observing major problems with the current release, they would rather continue operating it. New versions can create new problems (and most likely contain new errors and flaws).

Mini-implementation project

If the customer decides to license the new release, a mini-implementation project may be set up before the arrow back to the operation stage in figure 7-5 is followed. Sometimes a new release is accompanied by so many changes and extensions that it comes close to a new system. In such a case, user training and more preparation may be required before the system can go into operation. This is the reason for a second arrow going back from the "system upgrades" to the "preparing for transition" stage in figure 7-5.

Implementing an upgrade is a business project

The effort to successfully upgrade a standard package is often underestimated. One misconception is to think of implementing an upgrade of a business information system as an IT task. In practice, it is more a business project than an IT project, because the business side is responsible for determining the business case for the upgrade. Upgrading an ERP system, for example, requires that the business units establish the timetable for planning, installing and testing the upgrade to minimize disruptions to business processes. In a practical ERP upgrade project cited by Beatty and Williams as an example, it was found that the manufacturing department absorbed 43 % and the finance department 12 % of the project hours [Beatty 2006, p. 108].

7.2.3 A Vendor-specific Process Model

Implementing a large standard-software system such as an ERP or a CRM system is a voluminous task, including a large number of fine-grained activities, in particular for customization of the system. The implementation crew can easily get lost in the complex network of things to be done. Large software vendors have developed computer-supported tools to assist the organization in the implementation work. Such tools are based on a specific process model provided and followed by the vendor.

ASAP (Accelerated SAP)

A well-known example is SAP's process model for implementing ERP, supported by the ASAP (Accelerated SAP) toolset. ASAP was created in 1997 and based on worldwide experiences SAP consultants and user organizations had gained in ERP implementation projects. As the name suggests, the motivation for ASAP was to speed up lengthy implementation projects so that organizations could benefit from the ERP system "as soon as possible."

ASAP is not a process model but actually a toolset for ERP implementation, organized along the stages and activities on the road to a running system. A so-called "roadmap" guides the implementation process. The roadmap is divided into five major stages. All activities within the stages are supported and monitored by an automated implementation guide. ASAP provides detailed project plans in MS Project format to assist each stage.

ASAP roadmap

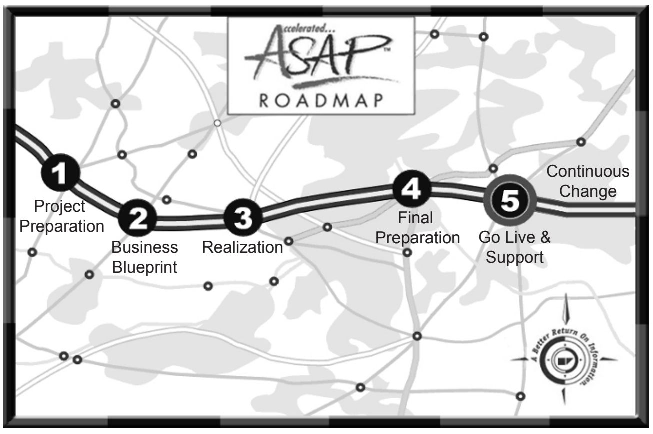
Although not explicitly named a process model, the ASAP roadmap indeed serves as one. Its major stages and tasks are illustrated in figure 7-6. The initial stages of the above generic process model – vendor and product selection, negotiations, contract etc. (cf. figure 7-5) – are not included because SAP obviously supports only the implementation of their own product. Process stages covered by the roadmap are the following:

(1) Project preparation – planning the implementation project. This includes: defining project goals and objectives; clarifying the scope of the implementation; defining the project schedule, the budget plan and the order in which the modules will be implemented; establishing the project organization and relevant committees; and assigning resources.

Reference model

(2) Business blueprint – creating a so-called blueprint in which the organization's requirements in terms of business processes and organizational structure are outlined. A reference model is created that can serve as a basis for the next stage, realization. SAP provides a detailed generic reference model from which the implementation team can select, modify and include processes (and other components) into the organization's specific reference model.

Figure 7-6 **ASAP roadmap [SAP 2005c, p. 2]**



© SAP AG

Different views of the IS

A complete reference model consists of various views of the information system: a process view, a functional view, an organizational view, a data view, an information-flow view and a communication view. All views are represented by graphical diagrams. For example, processes are represented by event-controlled process chains (EPCs) and functions hierarchies by decomposition diagrams.

The blueprint document also contains specifications of forms, reports and access rights that have to be realized in the next stage. The business blueprint stage is a very important one because it determines what the final system will be like. A "question and answer database" is provided to support the team in asking the right questions and not forgetting important issues in preparing the business blueprint. An automatically generated "business process master list" (BPML) can be used as a documentation of the answers.

(3) Realization – implementing the reference model defined in the business blueprint by configuring the selected modules. Customizing (cg. section 7.3) is the main task in this stage. First a so-called baseline system is configured, covering about 80 % of the blueprint's business processes. Special cases, individual requirements and exceptions making up the remaining 20 % are solved in the second phase, leading to the final configuration. Other important tasks of this phase are conducting integration tests and developing end-user documentation.

Implementation guide (IMG)

The major tool provided for customization is the *implementation guide (IMG)*. This tool helps the implementation team customize the system step by step, according to the requirements defined in the blueprint and/or with the help of the question and answer database and the business process master list (BPML).

(4) Final preparation – completing testing (in particular interoperability and performance testing), end-user training, system management and cutover preparations. All open issues should be resolved to ensure that all prerequisites for the system to go live have been fulfilled. Integration with other information systems and data cutover must be completed.

(5) Go live and support – moving from a pre-production environment to the live system. This stage includes activities such as production support, monitoring system transactions and system performance, tuning and/or removing performance bottlenecks, and removing errors.

Consulting firms specialized in SAP ERP implementation

ASAP is not the only roadmap for SAP ERP implementation. Many consulting firms have specialized in SAP ERP implementation and created their own roadmaps based on their own experience. SAP also provides more roadmaps, e.g. ASAP Focus for mid-size companies and Global ASAP for ERP implementation on a global scale. SAP's comprehensive application management platform (called "solution manager") contains ASAP tools as a part of the total functionality [SAP 2008].

7.3 Customizing Standard Software

Customizing standard software means adapting the software to the individual needs of a customer. Various methods, techniques and tools for customization exist, ranging from setting parameters, without touch-ing the program code, to writing and including individual programs. In the following sections, two types of customizing standard software are discussed: adjusting and extending the software.

7.3.1 Adjusting Standard Software

Setting parameters and selecting from options

Most standard business software can be adjusted by setting parameters or selecting from lists of options. Vendors of large software systems increasingly design their systems in such a way that they incorporate a lot of functionality – much more than an individual organization needs. Alternative functionality for the same problem is also provided so that a variety of customer organizations can select the solution which matches their specific requirements best.

No additional programming required

The final system operating at a customer's site is then configured (or generated) according to the specific parameter settings and selections. No additional programming is required. This is a major advantage for the customer, taking into consideration that developing new or changing existing code is expensive.

Using the SAP implementation guide

An example of adjusting an ERP system through parameter settings is given in the figures 7-7 to 7-9. In this example, the above mentioned *implementation guide* is used to configure an *SAP ERP* system. This tool guides the implementation staff through the customization steps. Three categories of parameters are distinguished:

– General settings, such as country, currency, calendar, time zones, measurement units etc.
– Settings regarding the organizational structure, such as factories, company codes, clients
– Module-specific settings, such as certain parameters for requirements planning within the materials management module.

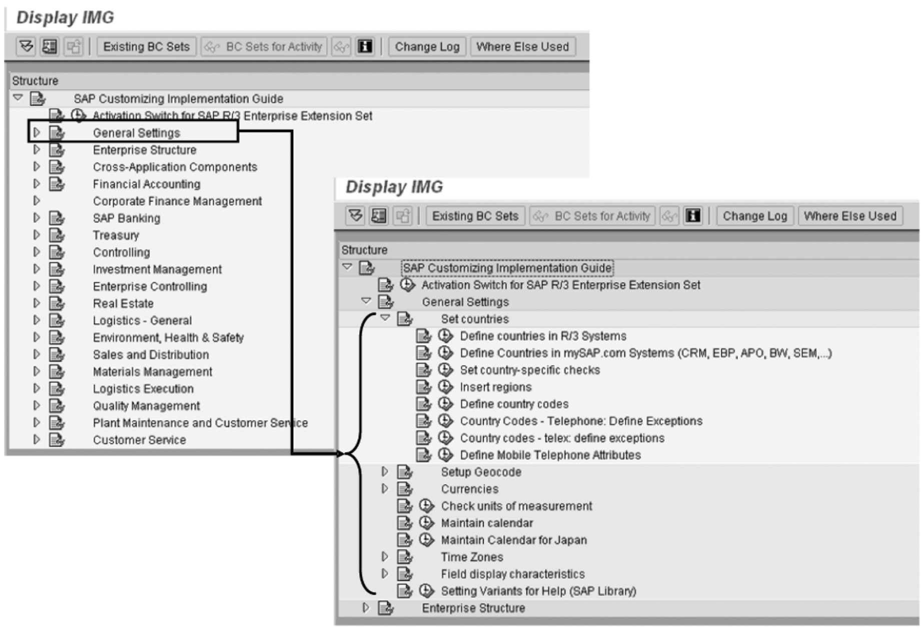
Customization areas

Figure 7-7 shows in the window on the left a screenshot with part of the navigation structure provided by the implementation guide. It illustrates the areas in which customization features are available. The substructure for general settings is expanded in the window on the right. The organization can specify country-specific parameters such as country name, telephone codes, currencies and time zones. In figure 7-8, the window to set the company's country is displayed.

Parameters for availability checks

Setting a module-specific parameter is illustrated in figure 7-9. The underlying problem is how exactly to perform an availability check for a particular material. Suppose this material is required for an upcoming production step or a customer order. The simple question: "is this material available?" can have many answers, depending on what type of stock is included in the check and which point in time the check refers to.

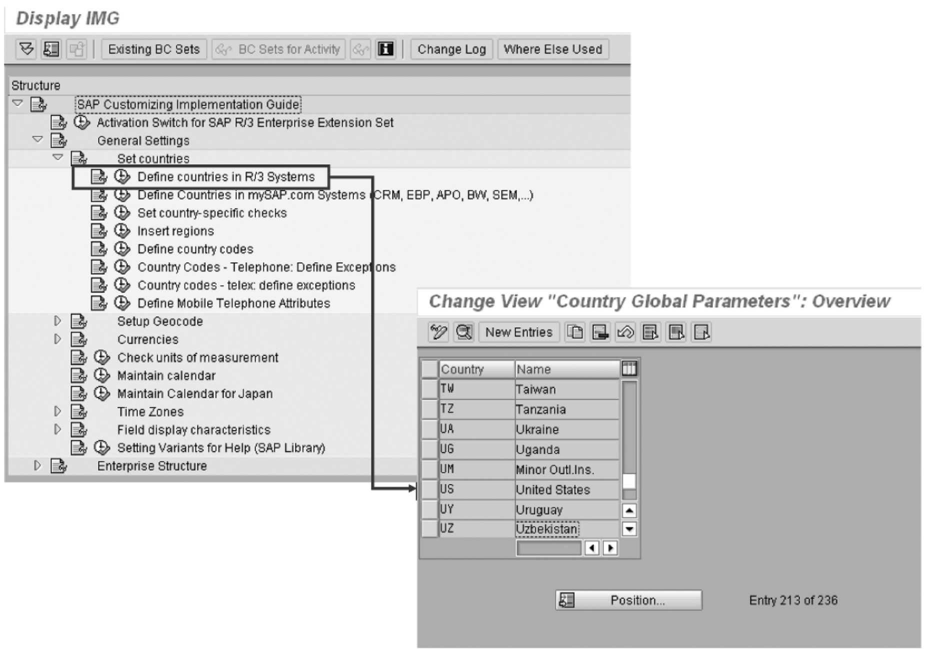
Figure 7-7 SAP implementation guide (IMG)


Display IMG
Existing BC Sets
BC Sets for Activity
Change Log
Where Else Used
Structure
SAP Customizing Implementation Guide
Activation Switch for SAP R/3 Enterprise Extension Set
General Settings
Enterprise Structure
Cross-Application Components
Financial Accounting
Corporate Finance Management
SAP Banking
Treasury
Controlling
Investment Management
Enterprise Controlling
Real Estate
Logistics - General
Environment, Health & Safety
Sales and Distribution
Materials Management
Logistics Execution
Quality Management
Plant Maintenance and Customer Service
Customer Service
Display IMG
Existing BC Sets
BC Sets for Activity
Change Log
Where Else Used
Structure
SAP Customizing Implementation Guide
Activation Switch for SAP R/3 Enterprise Extension Set
General Settings
Set countries
Define countries in R/3 Systems
Define Countries in mySAP.com Systems (CRM, EBP, APO, BW, SEM,...)
Set country-specific checks
Insert regions
Define country codes
Country Codes - Telephone: Define Exceptions
Country codes - telex: define exceptions
Define Mobile Telephone Attributes
Setup Geocode
Currencies
Check units of measurement
Maintain calendar
Maintain Calendar for Japan
Time Zones
Field display characteristics
Setting Variants for Help (SAP Library)
Enterprise Structure

© SAP AG

As the figure shows, stock that might be examined can be safety stock, stock in transfer, stock in quality inspection, stock that is just blocked for other orders, etc. In addition, availability depends on movement inwards and outwards from the inventory locations, e.g. on purchasing orders for the material and when these orders will be delivered. Material that is not available today but on the day when it is required in production may be regarded as "available". A replenishment lead time may need to be considered, because purchasing material takes some time.

Figure 7-8 Defining the country with the IMG



Display IMG

Structure

- SAP Customizing Implementation Guide
  - Activation Switch for SAP R/3 Enterprise Extension Set
  - General Settings
    - Set countries
      - Define countries in R/3 Systems
      - Define Countries in mySAP.com Systems (CRM, EBP, APO, BW, SEM,...)
      - Set country-specific checks
      - Insert regions
      - Define country codes
      - Country Codes - Telephone: Define Exceptions
      - Country codes - telex: define exceptions
      - Define Mobile Telephone Attributes
    - Setup Geocode
    - Currencies
    - Check units of measurement
    - Maintain calendar
    - Maintain Calendar for Japan
    - Time Zones
    - Field display characteristics
    - Setting Variants for Help (SAP Library)
  - Enterprise Structure

Change View "Country Global Parameters": Overview

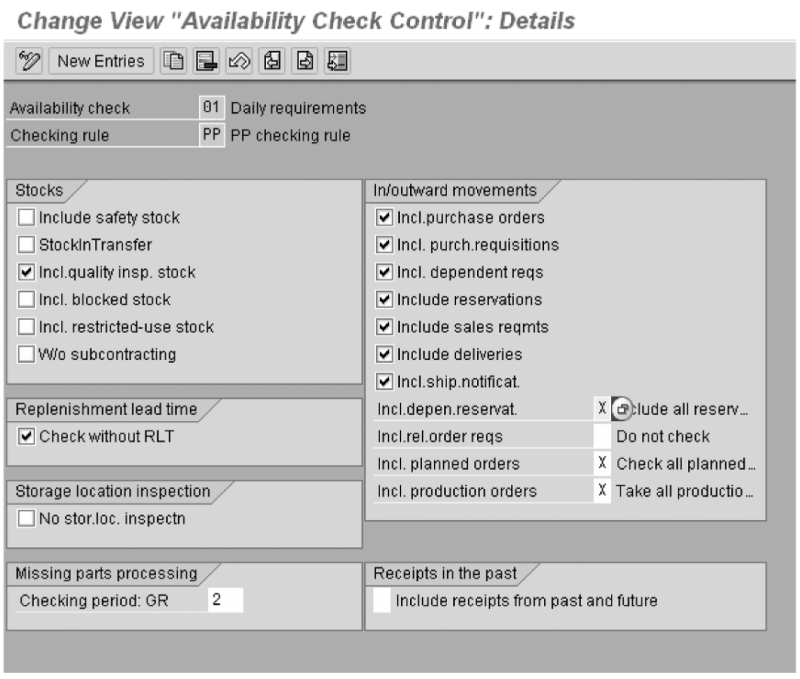
Country	Name
TW	Taiwan
TZ	Tanzania
UA	Ukraine
UG	Uganda
UM	Minor Outl.Ins.
US	United States
UY	Uruguay
UZ	Uzbekistan

Entry 213 of 236

© SAP AG

Another question is when to perform the check: when the production order is released, when it is opened or when it is saved? In SAP ERP, the customization team can choose among several options regarding the time of the availability check. Many more factors influence an availability check. While it is the production managers' responsibility to specify what to check and how to check it, it is the software that finally performs the check. Instead of writing a custom program that takes all user-specific requirements into account or modifying the standard-package code accordingly, parametrization makes it possible to satisfy the user's requirements and still leave the program code untouched.

Figure 7-9 Parameters for availability check



Availability check	01	Daily requirements
Checking rule	PP	PP checking rule

Stocks
- ☐ Include safety stock
- ☐ StockInTransfer
- ☑ Incl.quality insp. stock
- ☐ Incl. blocked stock
- ☐ Incl. restricted-use stock
- ☐ W/o subcontracting

Replenishment lead time
- ☑ Check without RLT

Storage location inspection
- ☐ No stor.loc. inspectn

Missing parts processing

Checking period: GR	2

In/outward movements
- ☑ Incl.purchase orders
- ☑ Incl. purch.requisitions
- ☑ Incl. dependent reqs
- ☑ Include reservations
- ☑ Include sales reqmts
- ☑ Include deliveries
- ☑ Incl.ship.notificat.

Incl.depen.reservat.	X	Include all reserv...
Incl.rel.order reqs		Do not check
Incl. planned orders	X	Check all planned...
Incl. production orders	X	Take all productio...

Receipts in the past
- ☐ Include receipts from past and future

© SAP AG

Too many parameters

Adjustment through parameter settings appears to be a cost-effective way to tailor standard software, making it an individual business solution. However, taking a closer look at parametrization reveals severe problems. These problems are not technical but business problems. There are essentially three reasons for the problems: the mass of parameters, the complexity of parameter interactions, and a lack of understanding of the business implications of parameter settings.

Too many parameter interactions

Consider the following example: The production-planning module of SAP ERP exhibits almost 200 parameters. Approximately 40 of them are related with materials (parts) managed in the MM module, similar to the ones discussed in the above examples. Imagine a small company with about 25,000 active materials. Then more than one million parameters have to be set [Dittrich 2006 p. 1]. Clearly this cannot be done manually. Therefore many parameters just remain untouched, and the initial default settings the system comes with are left as they are. Optimization potential is not exploited.

The second problem, *parameter interactions,* is due to the fact that the underlying business matters are interrelated. Setting one parameter in isolation can mean that the desired effect from setting a different parameter will not occur. Likewise, if the person setting a parameter does not understand the implications of this parameter on the business objectives, he or she will hardly be able to stipulate the parameter with a reasonable value.

7.3.2 Extending Standard Software: User Exits

When an organization needs functionality that is not provided by the selected standard package, this functionality may be developed separately and linked with the package.

User exit: an external program can be invoked

The so-called *user exits* (or program exits) are a traditional technique to extend standard software by custom modules. A user exit is a place in a program where an external program may be invoked. This program can be written by someone else, for example by the customer or by developers working on behalf of the customer (such as a consulting firm assisting the organization in the implementation of the package).

Providing user exits means that the vendor of the standard package deliberately opened the software for extensions. This is often the case when application-specific code is required that the standard-software vendor could not know in advance, or when that code is so specific that it would be different in each customer setting. If the required programming effort is significant, it does not pay for the vendor to develop many different code versions.

Extending standard software via user exits requires three steps:

Finding the correct user exit

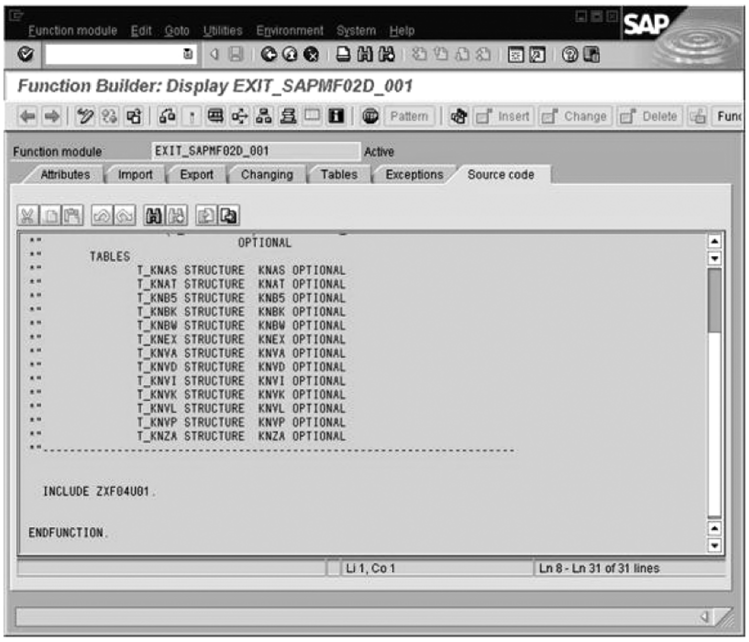
1. Finding the correct user exit in the source code of the standard package (provided that the source code is available) or in a code administration tool (provided that the software vendor offers such a tool). The documentation of a user exit should contain import and export specifications (parameter interfaces). Import means data input that the standard software provides for the program to be written. Export means data that the standard software will ac-

cept from the custom program. Finding the correct user exit may be a cumbersome step if user exits are not properly documented.

2. Writing the custom code. This includes processing import from and preparing output for export to the standard software.
3. Including the custom module in the standard software's source code. (This does not necessarily mean to physically embed it into the source code. It can also be loaded from a custom library at compile time or at runtime, for example.)

Including the custom program

Figure 7-10 User exit to include custom code [Becker 2007]



Function Builder: Display EXIT_SAPMF02D_001

Function module EXIT_SAPMF02D_001 Active

```
*"                        OPTIONAL
*"      TABLES
*"              T_KNAS STRUCTURE  KNAS OPTIONAL
*"              T_KNAT STRUCTURE  KNAT OPTIONAL
*"              T_KNB5 STRUCTURE  KNB5 OPTIONAL
*"              T_KNBK STRUCTURE  KNBK OPTIONAL
*"              T_KNBW STRUCTURE  KNBW OPTIONAL
*"              T_KNEX STRUCTURE  KNEX OPTIONAL
*"              T_KNVA STRUCTURE  KNVA OPTIONAL
*"              T_KNVD STRUCTURE  KNVD OPTIONAL
*"              T_KNVI STRUCTURE  KNVI OPTIONAL
*"              T_KNVK STRUCTURE  KNVK OPTIONAL
*"              T_KNVL STRUCTURE  KNVL OPTIONAL
*"              T_KNVP STRUCTURE  KNVP OPTIONAL
*"              T_KNZA STRUCTURE  KNZA OPTIONAL
*"----------------------------------------------------------------------


  INCLUDE ZXF04U01.


ENDFUNCTION.
```

Li 1, Co 1   Ln 8 - Ln 31 of 31 lines

© SAP AG

Software developed in a programming language such as Cobol, PL/1 or ABAP usually provides user exits in the form of code-includes or subprogram calls. Whenever an implementation makes use of a user exit, a custom procedure or function is called (and/or the included code is executed). Afterwards the standard module continues its work, possibly using data the subprogram exported.

Programming user exits is subject to constraints

Programming user exits is no fun. It is subject to constraints and conventions set by the standard software and by the underlying software technology. For example, extending a software package such as SAP R/3 which was written in ABAP means that the extension must also be programmed in ABAP. Variable, exit and subprogram names are given so that the customization programmer does not have much choice.

The screenshot in figure 7-10 shows an example from customizing the FI (financials) module of an SAP R/3 system [Becker 2007]. The programmer intends to write custom code for checking the content of a customer-data field. The user exit's name is "EXIT_SAPMF02D_001" (a typical naming convention in SAP's ABAP code).

The ABAP workbench makes the R/3 source code available where the custom code has to be entered. At the bottom of the screenshot, between the lines "INCLUDE ZXF04U01" and "ENDFUNCTION", the programmer will type the code extending the standard module[§]. This code will be executed by SAP R/3 whenever the flow of control in the FI module hits the user exit.

Event are often handled through user exits

User exits are often employed when *events* in the execution of the standard software occur. An event must be handled. However, the package vendor cannot always know in advance what a particular customer intends to do in the case of the event. Therefore event handling is often left to the customer.

7.3.3 APIs and the Hollywood Principle

User exits are "old technology"

User exits are "old technology", however still widely used. The reason is that user exits have been around since the early Assembler-language packages in the 1960s and 1970s. Basically the same mechanisms later found their ways into software written in third and fourth-generation languages such as Cobol, PL/1 and ABAP. Since SAP R/3 is based on

§ In the above example [Becker 2007], the following ABAP code was entered:

```
* User exit to ensure that all US customers have a group key
* entered on the customer master.
*
if i_kna1-land1 = 'US' and i_kna1-brsch = ' '.
  message e001(F2).
endif.
```

"old technology", programming user exits as shown above is still a common technique in the SAP users' world.

Most modern information systems are object-oriented. New systems are increasingly being developed in Java, as is the case for SAP's current developments. Under the object-oriented paradigm, user exits are out of place. Instead, object-oriented mechanisms are available to create extensions of existing software. Such mechanisms include abstract classes, interfaces, inheritance and polymorphism.

Object-oriented approaches

In order to apply object-oriented concepts in a systematic way, design patterns and other fundamental principles of object-oriented design can be employed. The so-called Hollywood principle and APIs realizing the encapsulation principle describe two approaches to connect standard modules with custom modules.

"Don't call us, we'll call you"

Custom extensions controlled by the standard package

The so-called Hollywood principle ("don't call us, we'll call you") applied to customization describes a form of collaboration in which the execution of custom extensions is controlled by the standard package. This means that the custom extension must be ready to be invoked, but it is under the control of the standard package to make it happen.

Conventional user exits as discussed in the previous section have actually always followed this approach. However, the name Hollywood principle became popular with modern software technology and the use of frameworks.

As described in section 5.2.2, a framework provides core functionality for a certain problem category through a set of abstract and concrete classes and interfaces, and it includes a mechanism to plug in additional classes and to customize and extend the provided classes. The framework specifies what the framework user has to provide. When a standard package was explicitly designed as a framework, then it is easy to extend because extensibility is built-in.

Customizing object-oriented software

Not only frameworks but also other object-oriented systems apply the Hollywood principle to allow for systematic extensions. A typical approach to permit individual organizations to *customize object-oriented software* is as follows.

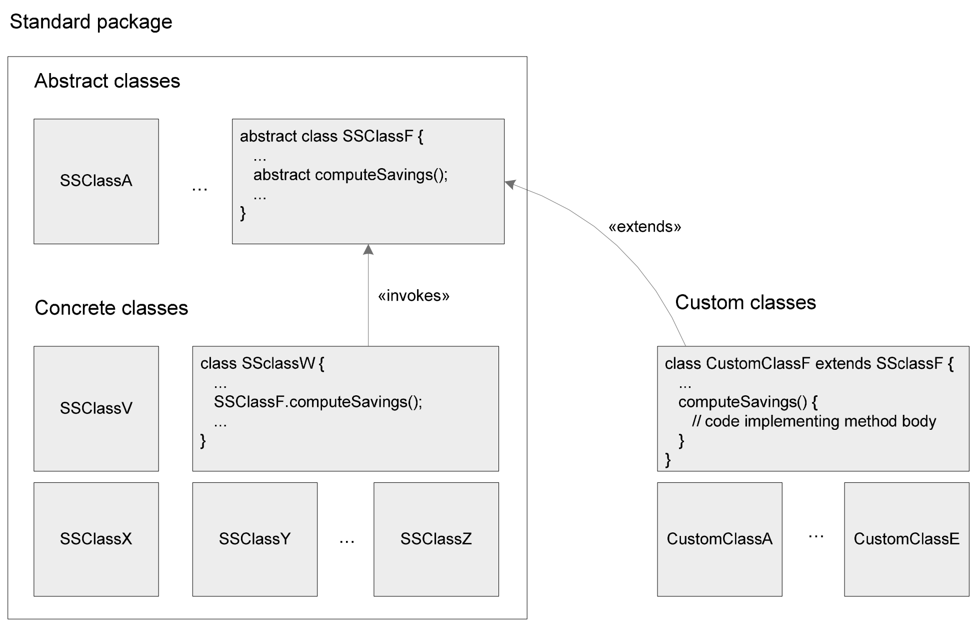
Abstract classes and interfaces

1. The software vendor provides abstract classes and interfaces containing little (or no) implementation code. The user organization subclasses the abstract classes and writes individual code. Similarly, the organization implements the interfaces so that they satisfy both the organization's needs and the software vendor's interface specification.

2. The standard modules invoke custom methods wherever they expect work to be done in a customer-specific way. This is possible because the vendor defined the signatures of these methods, either as abstract methods (i.e. methods of an abstract class) or as methods of an interface, and thus they are known in the standard package.

A Java-oriented schematic example is outlined in figure 7-11. The standard software on the left-hand side is composed of many classes, including some *abstract classes*. Abstract classes become concrete classes when the things not implemented in the abstract class (in particular methods) have been completely implemented.

Figure 7-11 Customizing a Java standard package



```
abstract class SSClassF {
   ...
   abstract computeSavings();
   ...
}
```

```
class SSclassW {
   ...
   SSClassF.computeSavings();
   ...
}
```

```
class CustomClassF extends SSclassF {
   ...
   computeSavings() {
      // code implementing method body
   }
}
```

This can happen both in the standard part of the system, performed by the vendor, and in the extensions part shown on the right-hand side, performed by the user organization. In the latter case, it is a customization.

Invoking customer-written methods that do not yet exist

Standard-software developers can invoke customer-written methods that do not yet exist at the time the standard software is created. This is possible because the same developers define the abstract methods as well (i.e. the signatures of these methods). For example, the standard-software class "SSClassW" can invoke the abstract "computeSavings" method which the customer will implement in the "CustomClassF" class. The class "CustomClassF" is a subclass of the abstract class "SSClassF". It inherits the standard features defined in the abstract "SSClassF" class and specifies in addition the custom features required inside the subclass.

Application programming interfaces (APIs)

Custom extensions can make use of APIs provided by the standard-software vendor. The underlying idea is similar to the parametrization approach: The software vendor develops program code for potentially needed functions in advance and makes it available to customers. However, while parametrization happens on the end-user level, APIs are used on the coding level.

APIs are interfaces to pre-written code

APIs are interfaces to pre-written code, usually collected in a library. The software vendor provides these interfaces, and the customer's developers can import the library code and invoke it through an API. Execution control is to some extent with the extension modules. The customer's developers write programs in which the pre-written standard-software code is invoked according to the flow of control in the extension module. This means that the programmer is relieved of writing code that the vendor's developers already produced before.

Most Java APIs are general-purpose APIs

Using APIs is a common approach in Java software development where thousands of APIs are available. The majority of APIs are general-purpose APIs in the sense that they provide useful programming functions that can be applied in many contexts. However, specialized and domain-specific APIs are also available.

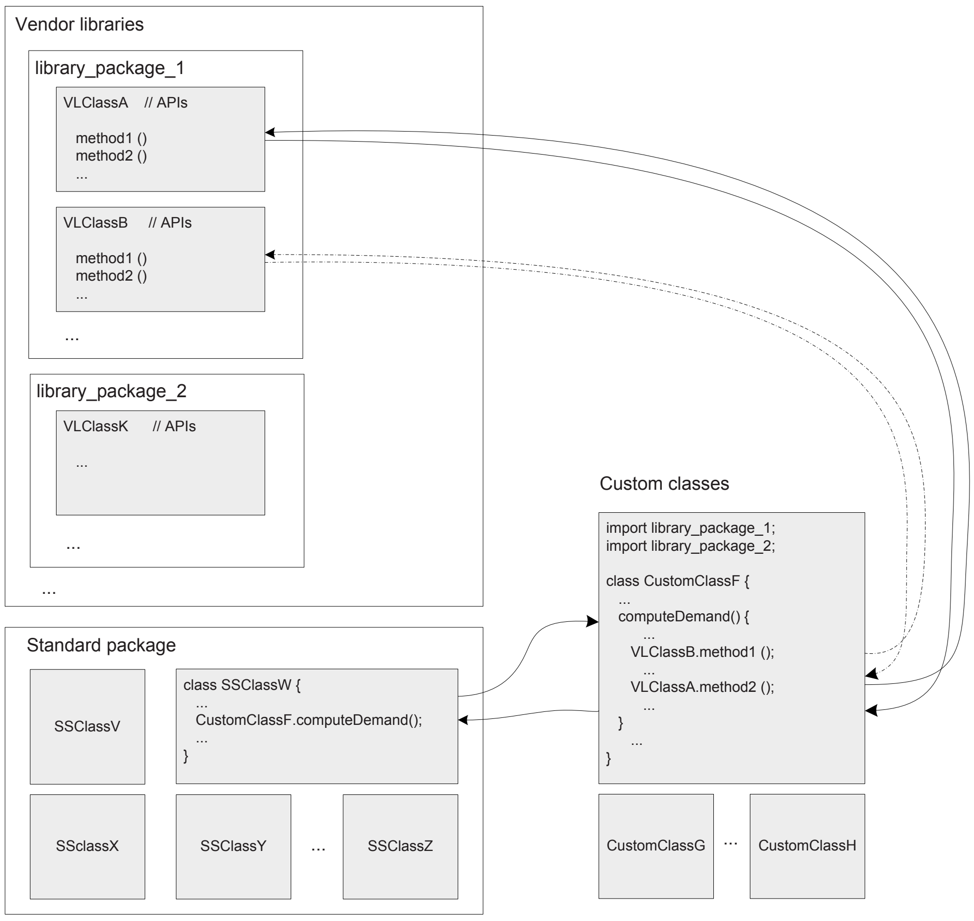
BAPIs (SAP's business APIs)

BAPIs (business APIs) are an example of business-oriented programming interfaces. These were introduced by SAP in the 1990s as interfaces to business functionality encapsulated in so-called business objects, e.g. "employee", "invoice", "product data" etc. [SAP1997]. Through the BAPIs, business objects embedded in SAP's software became available both to customers writing custom software and to software partners writing add-ons to SAP's software.

Using APIs is simple if the available libraries are known and understood. The only thing a custom developer needs to do is import the respective library package and invoke the methods provided. Figure 7-

12 illustrates this with a schematic example: The user organization has to create its own module for demand calculation because the software vendor's demand module does not reflect the organization's requirements.

Figure 7-12 **Flow of control using APIs**



What the vendor does, however, is to provide useful methods that simplify writing such a module.

The organization's individual demand-computation method is created within the "CustomClassF" class by a custom developer. This method, "computeDemand", calls two library methods from the "library_package_1" to accomplish its work. One is the method "VLClassB.method1" provided by the "VLClassB" class, the other one is "VLClassA.method2" provided by the "VLClassA" class.

7.4 Integrating Standard Software into the Information Systems Landscape

Old and new systems must coexist

Although a standard-software package may cover many business areas, it is never the only information system of the enterprise. Most likely other information systems, solving different problems than the new system, existed before and will continue to exist.

The new and the old systems are rarely completely isolated from each other but rather connected through common processes, data or workflows. This means that the new and the old systems have to collaborate in some way.

A common way of collaboration between different information systems is through *data*. If system A produces output that system B should process as input, and the formats of the data produced by A are different from the formats expected by B, then there is a compatibility problem that has to be solved.

Likewise, suppose both systems are employed within the same *workflow*. System A is an old Cobol program whereas B uses the latest Java technology. When a workflow step solved with the help of system A is completed, the next step may need the help of system B. Then the respective module of A, or some other mechanism, must trigger execution of the proper method of B, and data created in A must be available to B.

7.4.1 Enterprise Application Integration (EAI)

The problem of making heterogeneous systems work together is addressed in *enterprise application integration (EAI)*. The aim of EAI is to create integrated business solutions by combining existing information systems with the help of common *middleware*. This means that EAI allows autonomous information systems to share data, functions, objects and/or processes.

Integrating heterogeneous standard packages and legacy systems

Among other drivers, the growing dissemination of standard business software in the 1990s and the need to integrate it with existing information systems significantly stimulated research and development in EAI. Another driver for EAI was the need to integrate so-called *legacy systems*, i.e. stand-alone information systems based on "old" technology, with each other and with new developments (cf. section 7.5).

Despite its roots in standard-package and legacy integration, the scope of EAI is much wider. EAI also means that new solutions can be built on top of heterogeneous systems, leveraging earlier investments. Provided that up-to-date EAI technologies are available, EAI is an attractive foundation for the development of new solutions, because it requires neither big changes to existing systems nor extensive programming.

EAI simplifies the flow of information between departments using possibly heterogeneous information systems, improving internal processes in the enterprise.

Customers and suppliers can also benefit because EAI allows the company to manage relationships with customers and suppliers in an enterprise-wide integrated manner [Ruh 2001, p. 4]. Without integration, the customer (or the employee dealing with the customer) is often exposed to various stand-alone information systems. Each departmental system, for example, asks for information from the customer, which in many cases was given to another department's system before. With EAI, the point of contact for the customer (or for the employee dealing with the customer) appears to be *one* system. However, it is in fact the EAI technology that makes different stand-alone systems behave like one integrated system.

Middleware technologies

Middleware is application-independent software providing services that mediate between different application systems. In particular, middleware for EAI provides mechanisms to share information, to package functionality so that capabilities are accessible as services, and to coordinate business processes [Ruh 2001, pp. 2-3].

Common middleware technologies for enterprise application integration are message queuing and message brokers.

Two systems communicate through messages

Message queuing is an approach in which two systems communicate through messages. The sender system places a message in a queue for transmission. A queue manager forwards the message to a destination queue, and the destination queue holds the message until the recipient system is ready to process it [Cummins 2002, p. 6]. Coupling between the two systems is loose because the recipient need not be ready to process the message when it arrives. Furthermore, a transformation facility will convert the sender's data format to the format the recipient requires. A typical middleware for message queuing was IBM's *MQ Series* (now part of the WebSphere product suite and renamed to WebSphere MQ [Davies 2005b]).

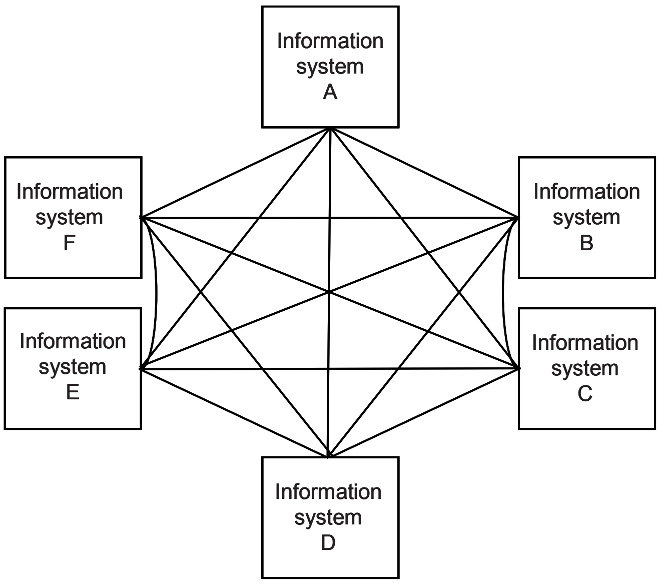
Message queuing is problematic when many systems have to be integrated

While message queuing is a useful approach for point-to-point connection between two systems, it is problematic when many systems have to be integrated in this way. As the number of point-to-point connections grows, this approach becomes inefficient. Figure 7-13 shows an example where n (n = 6) information systems are connected point-to-point. Since 15 double-ended connections exist, provisions for objects such as input queues, output queues and message channels have to be made 30 times, usually through program changes (i.e. declarations and calls to the message-queuing system in the 6 participating systems). The number of endpoints in application programs where changes have to be made is n * (n-1).

Message broker

A *message broker* is a facility that coordinates the communication in a network based on the exchange of formally defined messages. This concept is illustrated in figure 7-14. It allows information, in the form of messages, to flow between disparate applications and across multiple hardware and software platforms. A message broker receives messages from many sources and redirects them to their destinations. It delivers messages in the correct sequence and in the correct context of the destination system.

Figure 7-13 **Point-to-point connections in message queuing**



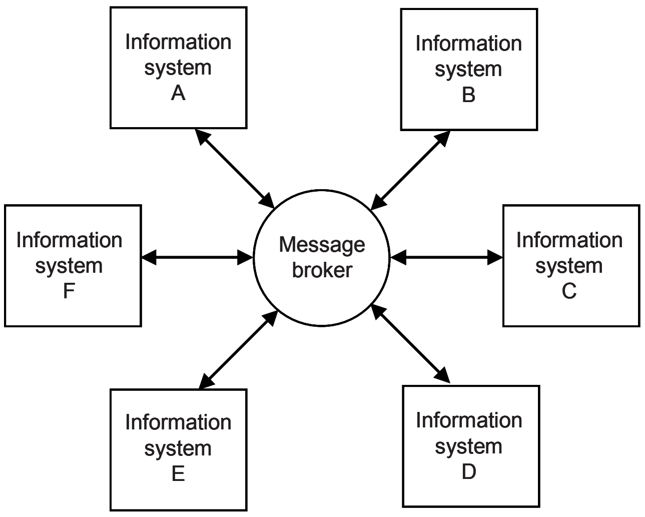
Translating and moving messages

A message broker transforms a message from the sender's format to the receiver's format (or more precisely, it translates a message from the sender's messaging protocol to the receiver's messaging protocol). In this way, message brokers are able to move messages from any type of system to any other type of system, provided that the broker understands the protocols used by these systems. Rules can be applied to the information that is flowing through the message broker in order to route, store, transform and retrieve the information. A well-known example of a message broker is IBM's WebSphere Message Broker [Davies 2005a].

Middleware topologies

An arrangement of nodes in a network is called a topology. Middleware connects a number of such nodes representing information systems. Topologies underlying the middleware include peer-to-peer, hub-and-spoke and bus topologies.

Figure 7-14 **Message-broker concept**



Peer-to-peer topology

– A *peer-to-peer (point-to-point) topology* connects each node with each other node. This is the case when individual connections between the information systems are established. Message-queuing systems as well as their predecessors, RPCs (remote procedure calls), implement a peer-to-peer topology. This was already illustrated in figure 7-13 above. State-of-the-art EAI approaches usually do not use a peer-to-peer topology because of the high integration effort.

Hub-and-spoke topology

– A *hub-and-spoke topology* has a central node onto which all other nodes are connected. In EAI middleware, the central node is the message broker. All connected information systems communicate with the message broker and not directly with each other. Figure 7-14 shows an example of a hub-and-spoke topology.

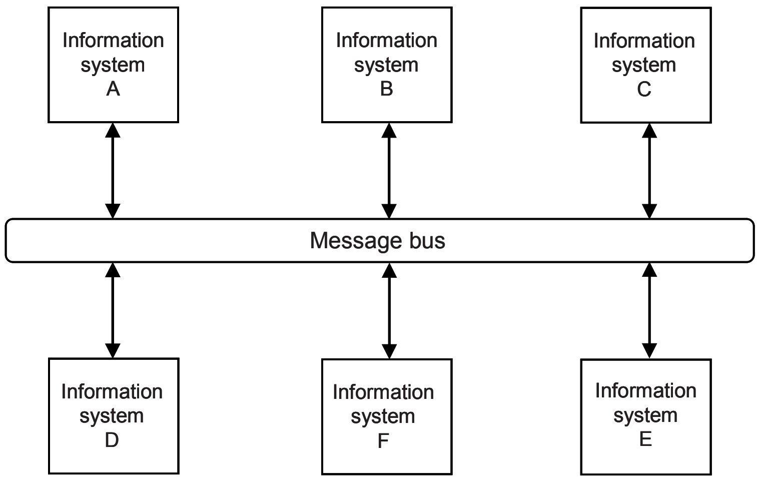
Bus topology

– A *bus topology* connects the nodes to a common transport component that controls and manages the communication between the nodes. A bus model allows two nodes to communicate while others have to wait until the communication is completed. In EAI, the bus implements the necessary integration mechanisms. Figure 7-15 illustrates the bus topology.

Enterprise service bus (ESB)

The bus topology has gained significant attention as a SOA component (SOA = service-oriented architecture; cf. section 3.3). When information systems functionality is organized in the form of services (e.g. enterprise services, web services) in a service-oriented architecture, a middleware such as an *enterprise service bus (ESB)* mediates between service requesters and service providers. An example of an enterprise service bus within a SOA was given earlier in figure 3-11 (chapter 3).

Figure 7-15 **Bus topology**



Integration levels

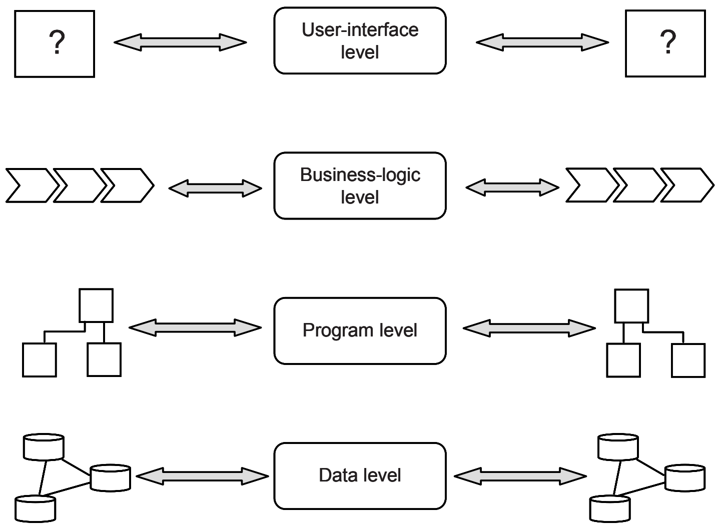
While middleware addresses integration on the level of software-technical infrastructure, users are more concerned with integration on the level of data, functions or processes. From this perspective, we can distinguish the following integration levels as illustrated in figure 7-16:

Data-level integration

➠ *Data level* – integrating different information systems in such a way that they can work with the same data. This means extracting information from one data source, perhaps transforming this information, and putting it in another database.

The advantage of data-level integration is that existing code remains largely untouched. Data-integration technology provides mechanisms to move data between databases and to transform data as necessary. Transformation of database schemata is a well-understood and mature approach. Still it requires significant effort, because in practice integration often includes not two or three databases but dozens or hundreds, with thousands of tables.

Figure 7-16 **Integration levels in EAI**



Transformation rules, mapping different semantics

Problems occur quite frequently. The main reason is that the database integrator and/or the end-user have to specify transformation rules. Often the rules are not obvious, and mistakes are made. Consider, for example, integration of a custom sales-force optimization program with the standard software's accounting module. Both databases have a "region" field, but in the first one regions are "north", "south", "west" and "east", while in the second one regions are "metropolitan", "suburbian" and "rural". It is obviously quite difficult to map these different semantics. Business knowledge is required so that appropriate rules can be specified.

Program-level integration

➠ *Program level* – integrating different information systems by allowing them to use each other's program functionalities. Application programming interfaces (APIs) help to do this, but only if they are available. Standard software usually provides APIs, as discussed in section 7.3.3. Conventional individual software sometime does, but mostly does not.

Changes to the program code are required in order to invoke the standard software's APIs. Furthermore, preparing an API invocation as well as results returned from the standard package may require processing by the individual software before and after the call. This means that program modifications and extensions are necessary.

Integrating two standard packages can be easy if the vendors considered this integration beforehand[§]. Otherwise it also requires custom programming in which the developer uses APIs of both systems.

Business-logic level

➠ *Business-logic level* – sharing business logic between different information systems. This means that the same business methods are accessible for all systems that need them, instead of having redundant (and sometimes inconsistent) implementations of the same method in these systems.

Consider, for example, an S&D (sales and distribution) system and an MRP (material requirements planning) system both calculating order-completion dates. If the two systems use different scheduling methods, then the MRP system will probably compute a different end date than the S&D system. As a consequence, the customer will be told the wrong delivery date, because the responsible sales representative works with the S&D system, but manufacturing follows the MRP system.

Business-logic level integration is substantially enabled by service-oriented architectures where business methods are available as enterprise services or web services (cf. sections 3.3.1 and 3.4.1).

User-interface level ("screen scraping")

➠ *User-interface level* – bringing information systems together via their user interfaces. This is usually done when the systems are so heterogeneous that they cannot be integrated otherwise, for example two legacy mainframe systems. The technique is also known as "screen scraping" because it takes mainly the screen input and/or output of the respective systems and integrates the input or the output on the user interface.

§ This is not unusual in a business environment. For example, vendors of specialized software often provide interfaces to SAP's ERP software, because they know that many of their customers will run an SAP system.

7.4.2 Patterns for Enterprise Application Integration

Since most organizations have an EAI problem, many approaches have come into existence and best practices have emerged. Like in other areas, documented *patterns* are available in which the experience gained by many integration developers and software architects is combined.

Enterprise integration patterns

In their book on *enterprise integration patterns*, Hohpe and Woolf collected 65 patterns for EAI [Hohpe 2004]. An online description of the patterns can be found at http://www.eaipatterns.com. Hohpe and Woolf's pattern categories are:

- *Integration styles* – describing different ways in which systems can be integrated. All patterns in this category follow the messaging style, meaning that sending a message does not require both systems to be up and ready at the same time. This style is documented in the "messaging" pattern.
- *Channel patterns* – describing fundamental attributes of a messaging system. These patterns refer to the way a sender and a receiver can communicate through a common channel. Channel patterns include the "point-to-point channel", "publish-subscribe channel", "channel adapter" and "message bus" patterns.
- *Message construction patterns* – documenting the intent, form and content of the messages. The base pattern for this category is the "message" pattern describing how two applications connected by a message channel can exchange a piece of information.
- *Routing patterns* – explaining mechanisms to direct messages from a sender to the correct receiver. These patterns consume messages from one channel and republish the message to another channel. They represent specific types of the "message router" pattern. Examples are the "message broker", "aggregator" and "splitter" patterns.
- *Transformation patterns* – describing ways to change the content of a message, e.g. transforming the data format used by the sending system; or adding, deleting or rearranging data. The base pattern here is "message translator".

- *Endpoint patterns* – describing how a node (i.e. a participating information system) connects to a messaging system so that it can send and receive messages. "Messaging gateway", "event-driven consumer" and "message dispatcher" are examples of endpoint patterns.
- *System management patterns* – providing mechanisms to keep a complex message-based system running. Taking into account that thousands or millions of messages per day must be processed, the EAI solution has to deal with error conditions, performance bottlenecks and changes in the participating systems. "Message history", "detour", "smart proxy" and "test message" are system management patterns addressing these requirements.

7.5 Legacy Systems

In most organizations, a large number of old information systems are in use. It is common to call them *legacy systems*, because they do not conform to the latest software technology. However, there are a number of reasons why this pejorative term is not justified.

7.5.1 Characteristics of Legacy Systems

Many companies depend on legacy systems

The so-called legacy systems are often of inestimable value to the organization. Many companies depend on them to such a degree that they would have to stop operations if these systems failed for more than a few days.

From an academic software-engineering point of view, legacy software is full of problems and should be replaced by modern software as quickly as possible. The most serious problem with legacy software is that the cost of maintenance and adaptation can be extremely high. Practice reports indicate that many companies spend 60 - 80 % of their IT budgets for just maintaining and adjusting legacy systems.

Software evolution

Most business information systems undergo an *evolution* throughout their lifetime. Software evolution means continuous incremental changes to the software. These changes can be caused by removal of errors which are detected in the operation phase; software enhancements (i.e. code modifications and extensions resulting from new end-user requirements); measures to improve software quality; adaptation of the software to a new hardware or software platform etc. A system built 20 or 30 years ago is likely to rely on features of a target machine, operating system and utility programs that may not be supported any more. Generations of software developers and maintenance programmers may have written new or modified old code inside and around the system.

Software is aging

In addition, software is aging. Changing user expectations will eventually not be reflected by the software's functionality any more, nor can the software be modified indefinitely. The more changes made to the software, the more likely that there are unforeseen side effects that lead to errors. The system's reliability and performance are also decreasing. Incremental changes tend to alter or remove symptoms, changing the initial design incrementally. An initially clean and well thought-out software design, and thus the software quality, is slowly degrading.

Maintenance makes software systems grow

Another empirical observation is that instead of actually changing existing program code, maintenance programmers prefer to write their own (new) code, because the existing code is not well enough understood. Thus the size of the software grows. While they do remove errors and modify or extend what is required, many maintenance programmers do not document what changes to the program they made. This creates additional difficulties for future maintenance tasks.

Taking all these factors into account, the volatility of an information system eventually becomes so high that doing yet more changes to the system is considered too risky. The state of the system has to be frozen. Reengineering or substituting the system cannot be to put off any more.

7.5.2 Integrating with Legacy Systems

Integrating standard software with legacy systems can be extremely difficult. Legacy systems do not provide APIs, and often they even lack documentation. The less documentation about a legacy system is available, the more one has to rely on the system's behavior. One remarkable characteristic of most legacy systems is that even though the internals of the system are not known and understood in detail, the system is running and working. What can be observed is only its behavior.

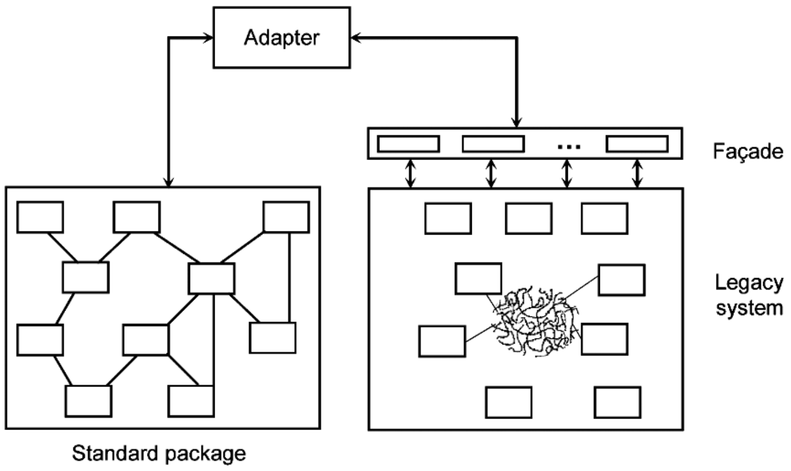
Façade pattern

Integration of a legacy system can be based on *what* the system does – not on *how* it does it –, taking the system more or less as a black box. A pattern that documents this approach is the *façade pattern* mentioned in section 5.2.2. This pattern provides an interface to one or more existing systems, hiding internal details of the system(s). The façade reflects behavior of the system(s) that should be accessible for other systems.

Depending on its software technology, a standard software system can interact with the façade in several ways. If the standard package is a conventional system providing user exits, then a user exit can be applied to invoke a façade method. More often the invocations have to be programmed inside the standard software, i.e. vendor code has to be modified so that it calls a façade method.

If the standard package is a modern object-oriented system, the façade could be implemented by subclassing superclasses provided by the standard software. However, this would mean that the legacy system is closely coupled to the standard software through the façade.

Figure 7-17 **Integrating legacy and standard software**



Adapter pattern

A better way is to define an application-independent façade and bridge the gap between the façade and the standard software's interface through an *adapter* (i.e. following the adapter pattern). Adapters are also called *wrappers* [Gamma 1995, p. 139]. Figure 7-17 illustrates this way of integrating legacy and standard software.

Infrastructural problems (e.g. disparate platforms on which the legacy system on the one hand and the standard package on the other hand are running) can prevent an immediate integration of the systems. Since the platform of the legacy system is likely to be an obsolete one, the legacy system may need to be migrated to a new platform. Migration requires that the legacy system can access the infrastructural services of the new platform. This can be accomplished by redesigning and recoding those parts of the legacy system that access infrastructural services – a rather awkward task, taking into consideration that the legacy code is not well understood.

Wrapping facilitates migration of a legacy system

A more manageable approach to making infrastructural services available to the legacy system is therefore the so-called *wrapping*. This means that the services the legacy system needs to do its work are provided via an interface, so that the legacy code can remain untouched.

7.5.3 Reengineering Legacy Systems

Advantages of legacy systems

Despite all the problems caused by old information systems, there are also serious arguments in their favor, including the following:

– Legacy software represents significant investments; developing or buying new systems would require large new investments.
– Experience and application knowledge have been gathered and incorporated into the software over the years. Generations of business people and information system developers have added business rules and processes, coded inside the legacy system's programs.
– In most organizations, no single person today has a complete understanding of the legacy system's internal mechanisms. Experience and knowledge are not explicitly recorded elsewhere, implying that they would also not be available for the development of a new system to replace the legacy system.

– Users are familiar with their programs. Errors have been removed over the years, or at least they are known. New programs contain new errors.

Integration of a legacy system with a new standard package, as well as integration with an information system developed inhouse, is a challenging problem. The main reason for this is that the integration effort usually requires changes in or extensions to the legacy code. For example, if a standard-package API has to be invoked, then the code calling the API must be written inside the legacy software. Or if the standard package provides useful input to the legacy system, additional code has to be written so that this input can be processed.

Developers need an understanding of the old system

In order to be able to modify a legacy system, developers need a sufficient understanding of the system – its design, coding and data structures. Unfortunately this understanding is very hard to obtain. Low software quality, opaque program structures ("spaghetti code"), missing documentation, redundant and inconsistent data (and even functions) are some of the problems. Often only the source code – or worse, machine code – from the legacy system is available. Before such problematic software can be modified in any way, some reengineering is required in most cases.

Business reengineering

The meaning of the term *reengineering* depends on the context. In business management, reengineering refers to the analysis and restructuring of business processes. In software engineering, the term reengineering comprises all activities aimed at the improved understanding and workings of the old software.

Software reengineering

Major stages of a *software reengineering* process are reverse engineering and restructuring:

Reverse engineering

– *Reverse engineering* focuses on deriving information of a higher abstraction level from low-level information. Reverse engineering has two particular sub-goals: re-documentation and design recovery. *Re-documentation* tries to accomplish what was neglected when the system was built: creating a documentation. *Design recovery* is an attempt to derive design models from code and data analysis. Examples of such models are operational program specifications, call hierarchies, functional models and data models (e.g. entity-relationship diagrams).

Restructuring, forward engineering

– *Restructuring* means shaping up program code, designs, specifications or concepts. Based on better structures, the legacy system can be improved through *forward engineering*. The goal of restructuring and forward engineering can be migration to a new platform, integration with other programs, or just improving software quality characteristics such as understandability and maintainability in order to reduce maintenance cost.

Integration-oriented reengineering

In the beginning, reengineering was primarily concerned with facilitating the maintenance of a legacy system or making the system ready to be ported to a new hardware or software environment. With the growing need for integrated systems, the focus shifted onto integration needs and the old system's interfaces with the outside (software) world. The term *integration-oriented reengineering* expresses this shift of focus.

A prerequisite for integration-oriented reengineering is understanding the old system in terms of its data structures and its functions. Even in writing a façade where the legacy code is left largely untouched (cf. section 7.5.2), an understanding of the internal structure is needed. When the legacy system has undergone reverse engineering and restructuring, the necessary steps to integrate it with other systems can be started.

Reverse-engineering process

Figure 7-18 illustrates the basic reverse-engineering process. It starts from the "highest" available abstraction of executable machine code. In the worst case, there is no higher representation of the program than the machine code and the database (or data files). Therefore the first step would be to create source code from the machine code. From the source code, and perhaps from knowledge of maintenance programmers and end-users, modules, screen definitions, data structures and call structures have to be derived. Eventually the level of data models and functional models may be reached. However, it is often not possible to derive these models because necessary information is missing.

Afterwards, restructuring and forward engineering can be started, with the aim of partially or totally reimplementing the design models (in particular, data and functional models) – provided that it was possible to derive a system design during reverse engineering. If not, then restructuring will be limited to cleaning up some of the code.

Reengineering repository

All representations and information derived in the reverse-engineering process are documented, preferably in a repository as outlined on the left-hand side of figure 7-18.

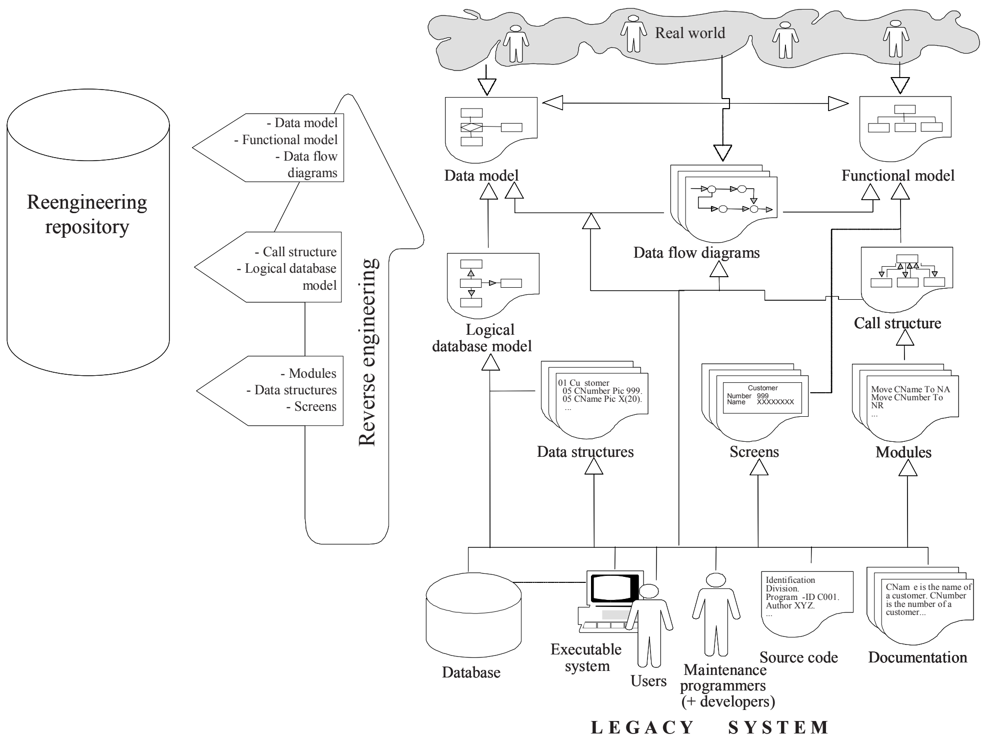
Automated tools for reverse engineering

Reverse engineering is supported by automated *tools* that create higher-level representations of a legacy system from lower-level representations. These tools include:

» Disassemblers and decompilers – producing source code from machine code.

» Model-capture tools – extracting information from the source code or from higher representation forms.

Figure 7-18 **Reverse engineering of legacy software**



» Analysis tools – helping to analyze and manipulate the extracted information.

» Documentation generators – creating condensed program information from source code, especially for Cobol programs. Java programs can be documented with Javadoc, a JDK component (cf. section 3.5.1).

» Visualization tools – creating graphical representations of information extracted from the legacy system, such as call graphs, program flowcharts and data-model diagrams.

8 Software Project Management

Information systems are usually developed in projects. Most tasks and activities discussed in the previous chapters take place within these projects. These projects must be planned, carried out, monitored and evaluated. Project management is the framework in which the planning, execution and controlling of projects occur.

Managing projects properly is of utmost importance. As mentioned in chapter 2, industry surveys report that only about 30 % of all application-software development projects are considered successful [Standish 2004]. Close to 20 % are failures (cancelled prior to completion or completed but never used), and the remaining 50 % are challenged (cost and/or time overrun, lacking features etc.).

In this chapter, we will first discuss project management issues in general and then with a special focus on information systems development. Although many project characteristics are the same, information system development exhibits particular properties.

8.1 Project Management Overview

Project management is an important field of management in many application areas. Not only information systems development but also many other undertakings with a focused goal take place in the form of a project. Examples are: building a bridge or an airport, preparing and conducting a cultural event, making a multi-million dollar movie, replacing the firm's IT infrastructure, and organizing a conference.

Project management has been studied for many years

Not surprisingly, project management has been studied in theory and practice and by various disciplines for many years – in business informatics as well as in management, computer science, manufacturing, construction and other disciplines. Many insights from various areas have been gathered, resulting in the emergence of a general body of project management knowledge.

However, it is not always justified to transfer practices gained in one problem domain to projects conducted in a different domain. While this mistake is often made, it has been identified as one of the reasons why ISD projects fail or why they do not produce the expected results. Therefore we will point out in section 8.1.3 the specifics of information systems development that distinguish it from other project types.

A vast body of literature exists

A vast body of literature about project management is available as well. It is not possible to condense all this gathered knowledge, based on many theoretical approaches and practical experiences, in one chapter of a book on making information systems. We will instead outline the major areas, methods, techniques and tools and refer readers interested in more detail to the relevant literature.

8.1.1 Tasks and Processes of Project Management

Discussing project management requires first an understanding of what a project is. From the many definitions of the term project that exist, we prefer the following one. It is an extension of the project definition given by the Project Management Institute (PMI) [PMI 2004, p. 5]:

Definition: project

> A *project* is a temporary endeavor undertaken to create a unique product, service or result through a series of related activities.

Relevant attributes which make an endeavor a project are "temporary", "unique" and "series of related activities":

– *Temporary* means that a project has a definite start date and a definite end. The project ends when its objectives have been achieved, or when the project is terminated for some other reason.
– *Unique* means that the outcome of the project exists only once, be it a material product, a service or another result. For example, a bridge is a unique outcome. Although many bridges have been built, each individual bridge is different. Likewise, each information system is different from every other information system although many systems have been created.
– A *series of related activities* is needed to complete the project. This characteristic refers to the temporal dimension of a project. Usually many activities – sequential and/or parallel activities – are required and have to be coordinated.

Project management deals with the various aspects of managing projects. The PMI defines project management as: "the application of knowledge, skills, tools and techniques to project activities to meet project requirements" [PMI 2004, p. 8]. This definition is adequate when "project requirements" are understood as including budget and schedule as well. On the other hand, the term "requirements" in ISD has a more focused meaning, in particular functional and non-functional requirements for the information system under development. Therefore we prefer Laudon's extension of the PMI's definition, which is [Laudon 2007, p. 557]:

Definition: project management

> *Project management* is the application of knowledge, skills, tools and techniques to achieve specific targets within specified budget and time constraints.

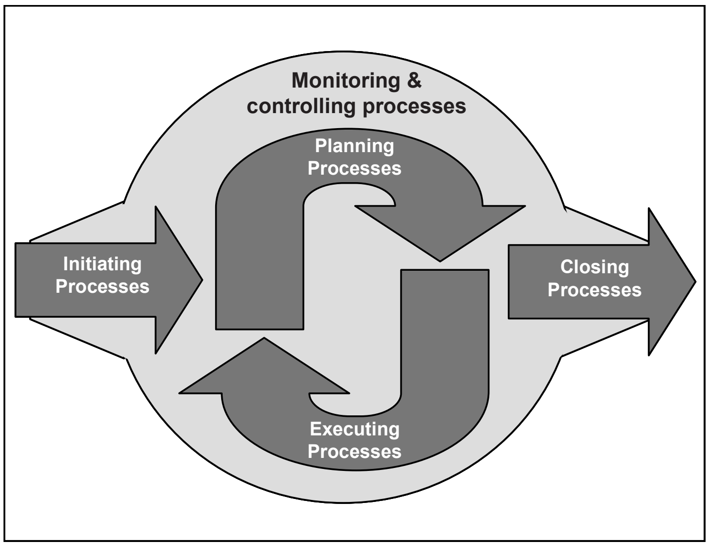
The starting point for project management activities is an approved project proposal as described in section 2.2.1, i.e. the decision to launch the project has been made. All project-management actions from then on are taken with the goal in mind to successfully achieve the project targets.

PMI project management processes

Project management involves sequences of activities that can be interpreted as processes. Figure 8-1 provides a graphic overview of the processes as defined by the PMI. These processes can be applied to the entire project or to a project phase. Processes cover initiating, planning, executing, monitoring and controlling, and closing a project (or a project phase) [PMI 2004, pp. 40-69]:

– *Initiating processes* facilitate the formal authorization to start a new project (or a project phase). This is often done by stakeholders outside to the project's scope of control, as discussed in sections 2.1 and 2.2.
– *Planning processes* support the planning of actions to attain the objectives and scope that the project was undertaken to address. The objects of the planning are so-called knowledge areas outlined further below.
– *Executing processes* have the goal to complete the work defined in the plan. Execution involves coordinating people and resources, as well as performing the activities of the project in accordance with the plan.

Figure 8-1 **PMI project management processes [PMI 2004, p. 40]**



– *Monitoring and controlling* processes regularly measure and monitor progress to identify variances from the project plan so that corrective actions can be taken when necessary.
– *Closing processes* formally terminate all activities of a project (or a project phase), pass on the completed product to others or close a cancelled project.

Activities which are required to initiate a project were already described in chapter 2. Closing processes comprising activities such as documentation, delivering the result and contract closure are mainly administrative and will not be discussed in detail. The focus of this chapter is on the planning, execution, monitoring and controlling aspects of project management.

8.1.2 Project Management Topics

What are the matters to be managed in a project? Perhaps a software developer first thinks of analysis, design and implementation activities, a senior manager thinks of the cost, an end-user of the system's quality and a project manager of the people to coordinate. All of these are important topics of project management, and even more topics may need to be addressed.

PMBOK (project management body of knowledge)

A comprehensive list of project management areas is described in the PMI's "Guide to the PMBOK" ("project management body of knowledge" [PMI 2004]). A generally accepted subset of the PMBOK considered applicable to most projects was adopted by the American National Standards Institute (ANSI) as an ANSI standard (IEEE Std 1490-2003).

IPMA competence baseline (ICB)

Outside America, the International Project Management Association (IPMA) has defined and published a comprehensive collection of project management areas in the IPMA competence baseline (ICB). The ICB comprises fields of competences a project manager should possess, differentiating between

– technical (e.g. scope & deliverables, time & project phases),
– behavioral (e.g. leadership, results orientation) and
– contextual (project portfolio orientation, project implementation)

competence elements [ICB 2006].

Both IPMA and PMI have international chapters or member organizations disseminating the ideas and standards of the parent organizations. Like some other project management associations, they offer courses and provide certification for project managers.

PMI project management areas

The project management areas addressed in the Project Management Institute's "Guide to the PMBOK" are [PMI 2004]:

– *Integration management:* Due to many interrelationships, the various processes and activities in a project cannot be treated in isolation. For example, cost management has connections with time management and risk management. The objective of integration management is to coordinate the various processes and activities in a project.
– *Scope management:* The purpose of this area is to ensure that the project includes all the work required to complete the project successfully, but not more. Scope management is concerned with what is and what is not included in the project.
– *Time management:* A project consists of a large number of activities with diverse connections and dependencies. Time management is responsible for scheduling and timely completion of all activities and of the entire project. This will be discussed in more detail in section 8.2.1.
– *Cost management:* Since the total cost of a project is not known in advance, it has to be estimated. This was discussed in section 2.3.4. Costs then have to be broken down and assigned to work packages or activities, and controlled in the course of the project. When variances from the budget are detected, corrective actions have to be initiated.
– *Quality management:* All activities that determine quality policies, objectives, standards and responsibilities are summarized under quality management. This includes activities to plan, assure and control both the quality of the project's results and the quality of the project management processes.
– *Human resources management:* This area is concerned with identifying project roles and responsibilities, staffing the project, improving competencies and interaction of team members, tracking team member performance, resolving issues and coordinating changes of the team.
– *Communications management:* Generating, collecting, distributing, storing and retrieving project information in a timely and appropriate manner are crucial for project success. Processes and activities for

effective information exchange and reporting are included in this area.

– *Risk management:* Since projects can be exposed to significant risks, the management of these risks is of utmost importance. Risk management includes the identification and analysis of risks, the development of counter-measures, the monitoring and controlling of risks, and the evaluation of measures in response to the risks.

– *Procurement management:* Projects often require products, services or other results from outside. All activities and processes to purchase or acquire the products, services or results (e.g. contracting, supplier selection) are part of procurement management.

8.1.3 Special Aspects of ISD Project Management

Not everything is applicable to ISD

A substantial body of knowledge has been accumulated in the field of project management. However, not everything that is found useful in other fields is applicable to information systems development projects, at least not one-to-one. Most of the agreed and documented project-management knowledge was gathered in other areas, where circumstances and conditions are different from software development.

Applying an inappropriate project-management approach or relying on wrong assumptions about project matters can be sources of serious problems. Projects can fail if they are not managed properly. Therefore pointed out below are some important points which should be considered when a software-development project has to be managed:

Cost and time projections are less reliable

➡ *Certainty:* Cost and time management are more difficult because the data on which cost and time projections are based are less reliable. In other areas such as construction and manufacturing, where a good deal of project management knowledge has its origin, the required project activities are better known, more stable and easier to estimate. For example, we know pretty exactly how long it takes to apply a layer of concrete on a 50 m bridge on top of a steel girder infrastructure. However, we do not know with the same certainty how long it takes to implement a layer of software on top of a network infrastructure, because more things can go wrong or do not work as expected.

Activities are less repetitive

➡ *Repetitiveness:* One reason why activities can be predicted better in construction and manufacturing projects is that many activities occur in exactly the same or a similar way as they did in earlier projects. Even though the same typical activitites will be found in any software development project, factors influencing time and effort vary significantly. For example, to remove an error in integration testing can take half an hour, but it can also take a week.

When the organization's projects over time are very similar, more reliable estimates can be given. This is an assumption underlying the Cocomo II model discussed in section 2.4.3. Based on a series of earlier projects, an organization can calibrate the parameters of the estimation equations. In this way, they get reliable estimates, provided that the current type of project matches the previous projects.

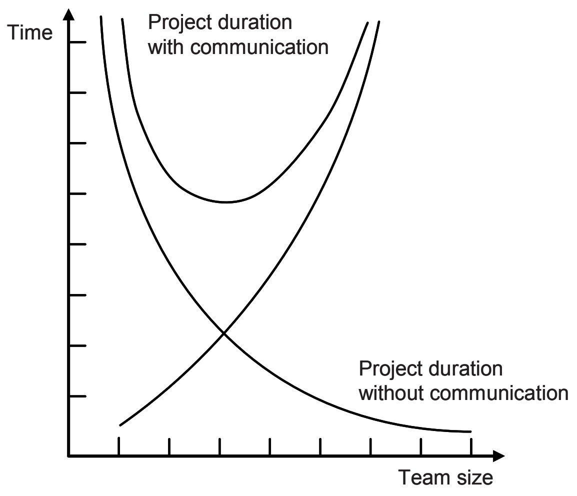
More communication needed if the problem is not clear

➡ *Communication:* Each software development project is different, not only regarding its scope and size but also its complexity and how well the problem to be solved is understood. The less clear the problem and its solution beforehand are, the more communication among the project team is required. Communication takes time, reducing the amount of time available for "productive" work. The larger the team is, the more team members usually need to communicate with each other. On the other hand, more people get the work done faster.

Figure 8-2 shows the effect of communication on productivity with the help of a schematic illustration. If no communication was needed, the project would end the faster the more people work on the team. However, project members need to communicate and additional time for their interactions must be considered, so the project takes longer. The communication effort increases progressively as the number of people involved increases. In fact, the figure suggests that there is an optimal team size from the perspective of communication. While this is true under idealistic assumptions, staffing a project in practice depends on more factors (e.g. skill requirements, availability, temporal aspects) than those underlying the figure. Nevertheless it becomes clear that communication needs have a significant effect on team size and project duration.

It should be noted that the curve for communication effort depends on the amount and intensity of communication required which in turn depend on factors such as how complex the system is, how well it is understood, and how experienced the team members are. Consequently the curve can also be further up or further down in the figure, resulting in different curves for project duration as well.

Figure 8-2 **Effect of communication on productivity**



Target shift

➡ *Moving target:* A frequent observation in practical software-development projects is that the system initially ordered by the customer is not the system finally delivered to the customer. This is primarily due to a "target shift" – by the customer, by the software organization developing the system or by both. A target shift occurs for many reasons, for example: the customer's market has changed since the system was ordered; a major competitor came out with a similar solution faster, leading to new requirements; new technologies have emerged, requiring a radical redesign; new stakeholders are in the game, with different interests; etc. The moving-target syndrome is a major reason for costs and deadlines overrunning.

Incremental scope changes

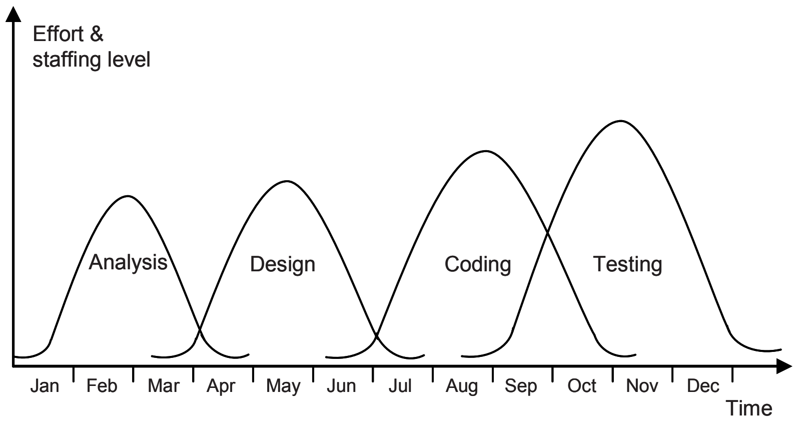
➡ *Scope creep:* A problem similar to the above scenario, scope creep is caused by the scope of the system under development undergoing small incremental changes. A superficial reason for this is that the customer's requirements are changing. Digging a little deeper, the actual reasons are twofold: 1) Requirements were not sufficiently clear in the beginning, when they were captured and documented in a requirements specification. As things become clearer in the course of the project, requirements need to be reformulated or changed. 2) When end-users and other stakeholders are involved in the development process, they get to see initial or intermediate solutions throughout the process. This helps them to understand what they will finally obtain, and at the same time they get new ideas about how things might be improved or extended by additional useful features. The more user involvement is realized, the more likely it is that requirements will change incrementally or that new requirements will emerge.

While adapting the scope of the project to the users' needs is good for the users, it requires more work on the developers' side that was not calculated before. The so-called scope creep is followed by a "cost creep" – a source of conflict between the customer and the contractor. The parties are forced to agree on what is actually covered by the initial requirements specification or contract, and what is not, i.e. what are the new requirements which need to be invoiced separately.

Inappropriate organization and allocation

➡ *Process model:* Many decisions in a software-development project depend on the chosen process model: the work plan, the project organization, roles, staffing and other responsibilities of project management. An often encountered mistake is to impose an inappropriate project organization onto the development effort.

Figure 8-3 **Work distribution scheme**



Another problem is insufficient consideration of timely staff allocation, i.e. assigning certain skills when they are needed. The example of a common work distribution scheme shown in figure 8-3 illustrates this point. The peak demand for analysts in this example is in February - March, while testers are mostly needed October to December.

This work distribution and staff allocation can work well when the project follows a waterfall model. However, when the project manager decides to apply an iterative approach because it appears better suited to the problem at hand, an allocation of human resources according to the scheme in figure 8-3 would be completely inappropriate. In an iterative model, testers will be needed in each cycle (e.g. every three months), and the same is true for analysis experts. If the upper management allocates the project's staffing "as it is always done", then the project will get into trouble!

8.2 Project Planning

Some of the planning problems addressed in the project management areas outlined in section 8.1.2 have already been discussed in previous chapters. For example, the project scope is decided upon in requirements engineering; the planning of project cost, in particular cost estimation methods, were described in section 2.3.4; project risks were addressed in chapter 2.

Quality management is relevant to all ISD phases. It was also addressed in various contexts before. For example, quality issues are explicitly treated by one of the RUP (Rational unified process) best practices – "focus continuously on quality" (cf. section 4.3.3). Communications management together with tools supporting it will be discussed in chapter 9.

In section 8.2, we focus on two important areas of project planning: 1) work and time planning on different levels of detail, including the scheduling of activities; 2) planning the project organization.

8.2.1 Activity and Time Planning

The aim of activity and time planning is to define the activities required to accomplish the project result, to bring the activities into an appropriate sequence, to determine the resources and the time needed to perform an activity, and to create a schedule based on an analysis of the activities and their relationships. The PMBOK distinguishes several processes for activity and time planning (plus one process to control the schedule) [PMI 2004, pp. 123-156]:

Activity definition

Work breakdown structure (WBS)

The overall work to be done in the project has to be decomposed into manageable units. Complex tasks are broken down into subtasks and further refined. The result is called a *work breakdown structure (WBS).* (In the PMBOK, the WBS is actually created under the label "scope management", and the work units on the lowest refinement level are called "work packages.")

Deriving activities from the WBS

Typical approaches to derive activities are hierarchical decomposition and the use of templates. *Hierarchical decomposition* means breaking coarse-grained work units down into fine-grained work units (e.g. work packages into activities). However, when the project structure is not completely new, it may not be necessary to do the same decomposition work again that has already been done in previous projects in the same way. In such a case, activities can be adopted from a previous project or from a *template* in which activity definitions of previous projects have been gathered.

Activity definitions include activity descriptions and attributes such as predecessor and successor activities, logical relationships, constraints and assumptions. They have to be documented, for example in an *activity list*.

Activity sequencing

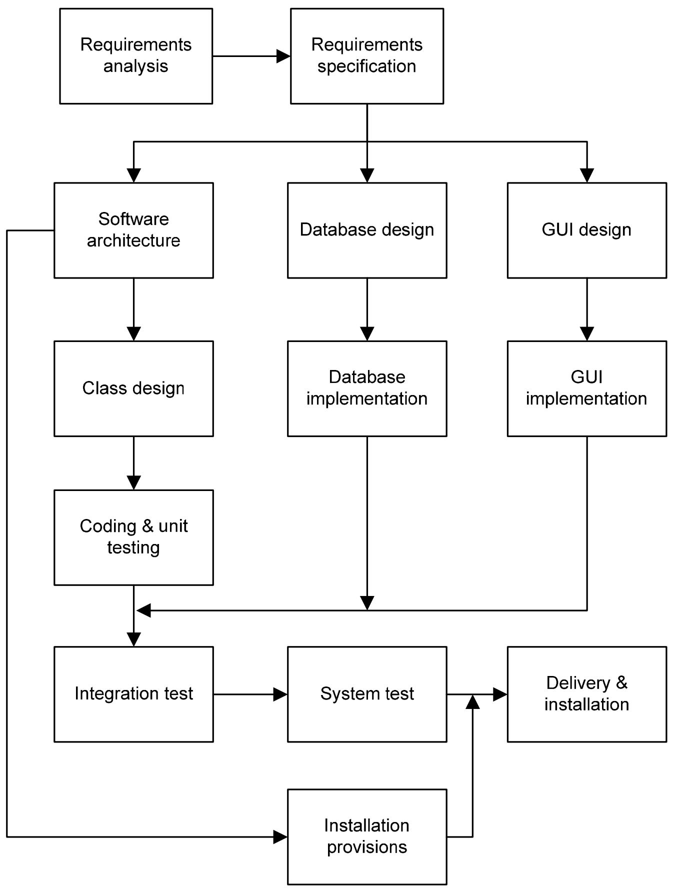
Precedence graph

Evaluating the logical relationships between activities, especially predecessor and successor relationships, and arranging the activities in a logical structure is the goal of activity sequencing. The result is a *prece-*

dence graph, showing which activities precede and/or succeed other activities.

An example is given in figure 8-4. The figure contains 13 activities and their logical relationships. Arrows indicate successor-predecessor relationships. The activity "database implementation", for example, requires that the activity "database design" has been previously completed. As the figure shows, some activities have more than one predecessor or successor, and some activities can be performed in parallel.

Figure 8-4 **Activity precedence graph**



Precedence-relationship types

Several types of precedence relationship are possible. The most common one – and also the one assumed in figure 8-4 – is that the preceding activity must be finished before the successor can start (*"finish-to-start"*). For example, testing of module A must be finished before integrating A with other modules can start. Other types are [PMI 2007, p. 132]:

– "Finish-to-finish" – the completion of the successor activity depends on the completion of the predecessor activity.

– "Start-to-start" – the initiation of the successor activity depends on the initiation of the predecessor activity.

– "Start-to-finish" – the completion of the successor activity depends on the initiation of the predecessor activity.

Activity-on-node network

The type of diagram in figure 8-4 is called an *activity-on-node* network because the nodes represent activities. Usually the nodes of an activity-on-node diagram contain more information than just activity names. This will be illustrated below in figure 8-7.

Activity-on-arrow network

A different type of diagram, called an *activity-on-arrow* (or activity-on-arc) network, is obtained when the activities are represented by arrows (arcs). The nodes of such a network stand for the start and the end of activities and are regarded as events. Figure 8-5 shows an example of an activity-on-arrow network.

The nodes of this network contain four types of information: 1) node number (top sector), 2) earliest end (left sector), 3) latest end (right sector), and 4) float (i.e. time buffer; bottom sector). For example, 5-11-19-8 in the upper center node means that the state of the network after performing activity D is that the earliest end of D is 11, the latest end of D is 19, and thus the buffer is 8.

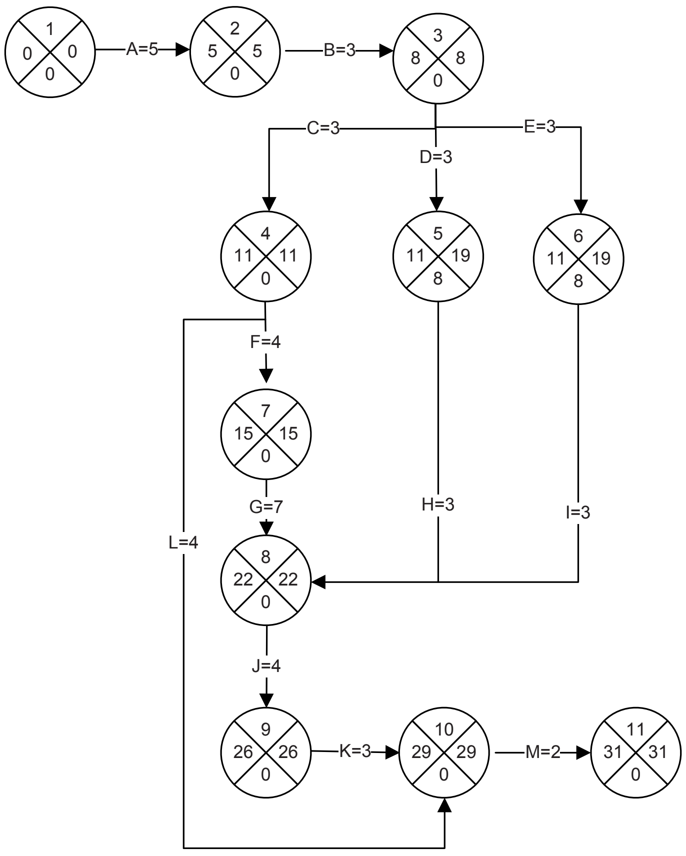
Information on the arrows refers to activities. For example, D=3 means that activity D (i.e. "database design", cf. figure 8-6) has a duration of 3 time units (e.g. weeks).

Activity resource & duration planning

Resources

Determining what resources and what quantities of each resource are needed to perform each project activity, and when the resources will be available, is the goal of *activity resource planning.* Resources can be persons, skills, computing equipment, IT infrastructure etc. When the demand for resources is uncertain, it must be estimated, for example using expert judgment.

Figure 8-5 Activity-on-arrow network



Activity durations

Activity durations are needed for scheduling and any other time-related project planning. However, they are often the most difficult estimates to determine, and at the same time the most critical ones. Project failures and deadline violations are often due to unrealistic assumptions about activity durations. This problem is particularly serious in software-development projects. In many work areas of such projects, it is quite difficult to predict activity durations with a sufficient degree of certainty.

Three-point estimation

Techniques for estimating activity durations include the application of expert judgment and analogies from previous projects. A so-called *three-point estimation* can be used when uncertainty is high, leading to "most likely", "optimistic" and "pessimistic" estimates.

Schedule development

Schedule development methods

An activity precedence graph (either an activity-on-node or activity-on-arrow graph) and activity durations are the most important inputs to activity scheduling. Availability of resources must also be taken into account. The goal is to develop a schedule in which planned start and end dates are assigned to all activities. Methods supporting schedule development are based on graph theory, especially network theory. Common methods are:

– Critical path method (CPM)
– Metra potential method (MPM)
– Program evaluation and review technique (PERT)
– Graphical evaluation and review technique (GERT)
– Critical chain method (CCM)

When a software tool for scheduling is available, the project manager basically needs to enter quantitative data from the activities' definitions (such as durations and predecessors). The tool then evaluates the interrelationships and creates a schedule. This will be demonstrated in section 8.2.3. Although scheduling is rarely done by hand, we will briefly explain the basic procedure here.

Critical path method (CPM)

Suppose an activity-on-node network as in figure 8-4 above was produced in the activity-sequencing step. Details of the activities are provided in figure 8-6. From these data, a schedule can be created. We will do so with the help of *CPM (critical path method)*. CPM calculates early and late start and end dates for all activities plus buffer times (so-called float):

– The *earliest* possible start dates are computed in a forward-pass analysis of the network, beginning with the earliest possible start of the first activity (e.g. "today").

– The *latest* possible end dates are obtained from a backward-pass analysis of the network, starting with the desired completion date of the last activity and calculating start and end dates of this activity's immediate predecessors.
– *Float (slack)* is the amount of time an activity can be shifted forward or backward without causing a conflict with any of its predecessors or successors. The sum of all floats on any path through the network (from the first activity to the last activity) is called the *total float*. It can be computed as the difference between the sum of all the earliest start dates minus the sum of all the latest start dates.

Figure 8-6 **Activity data (example)**

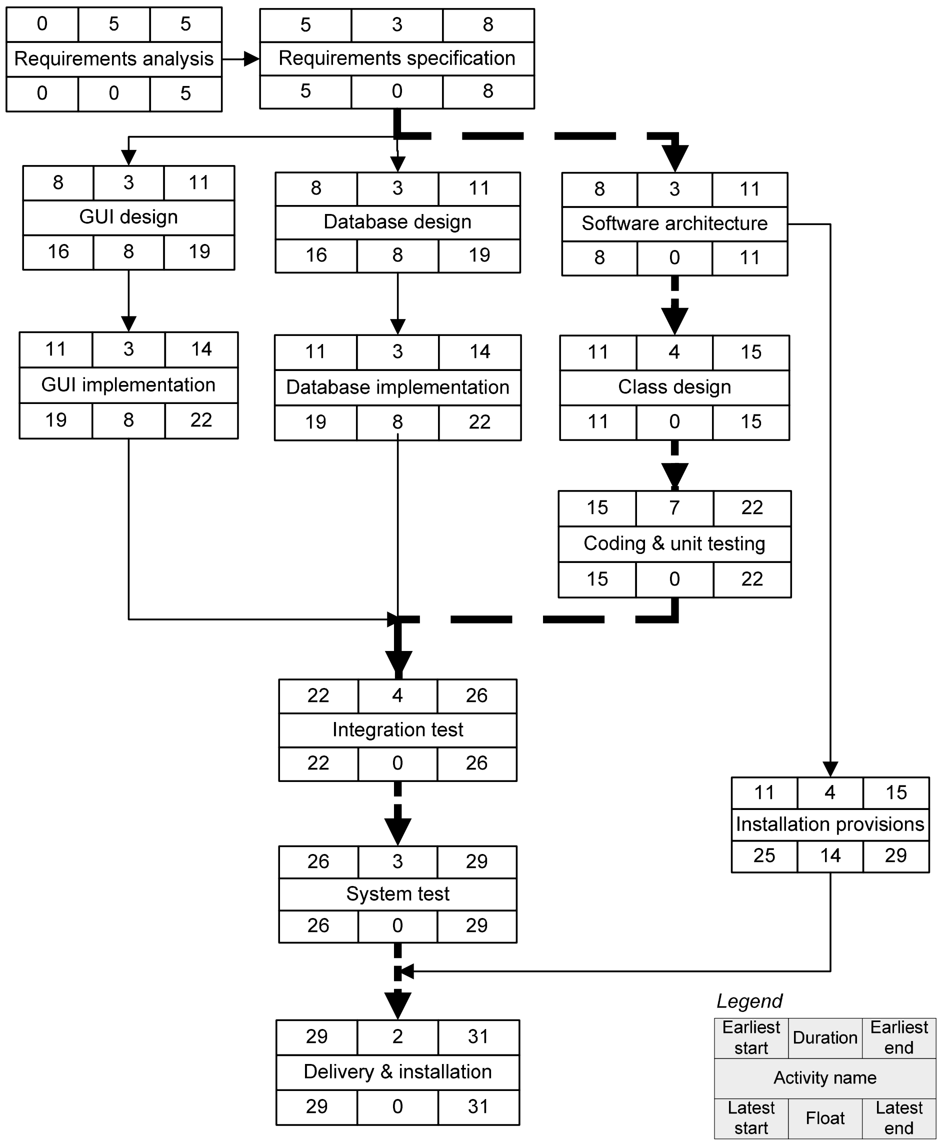
Activity ID	Activity Description	Predecessors	Duration
A	Requirements analysis	-	5
B	Requirements specification	A	3
C	Software architecture	B	3
D	Database design	B	3
E	Database implementation	D	3
F	GUI design	B	3
G	GUI implementation	F	3
H	Class design	C	4
I	Coding & unit testing	H	7
J	Integration test	E, G, I	4
K	System test	J	3
L	Installation provisions	C	4
M	Delivery & installation	K, L	2

Critical path

If a network path has a total float equal to or less than zero, it is called a *critical path*. Such a path is critical to the network because there are no time reserves. If anything goes wrong, the project completion date is endangered. Activities on a critical path are called *critical activities* because any delay in the completion time of the activity will cause the entire project to be delayed.

Figure 8-7 shows a network with 13 activities and a critical path. Activities connected by boldface dashed arrows are critical activities.

Figure 8-7 **Activity-on-node network with a critical path**



This can be seen from the time buffers (float) that are zero and the earliest and latest start and end dates which are identical. An activity such as "database implementation" is not critical because it has a float of 8 (= 19 - 11) time units (e.g. weeks). This means that it can start any time between weeks 11 and 19 as long as it is completed by week 22. This is the latest feasible completion date. Otherwise there would be a problem with the successor activity ("integration test") because this critical activity must start in week 22.

CPM can be combined with the above mentioned three-point estimation, using "most likely", "optimistic" and "pessimistic" estimates for activity durations. As a result, three schedules can be obtained, a most likely, an optimistic and a pessimistic schedule.

Alternatively, the three types of estimate can be weighted and combined to one *expected value* for each activity duration. Applying the CPM method based on expected values yields a schedule that has a certain probability attached to it. This probability can be computed from the expected values and their variances. It tells the project manager how likely it is that the project will be completed as expected.

CPM is a simple and an easy-to-use scheduling method. However, it has many limitations and drawbacks that make it unsuitable to model complex activity relationships and dependencies. More flexible, but not as easy to use are methods that consider stochastic activity durations and decisions in the process, such as MPM, PERT and GERT. The reader interested in more information should consult dedicated literature about network analysis and planning methods.

Schedule compression

Developing a feasible schedule with the help of any of the mentioned methods may not be possible. This can happen if the given activity durations are too long or the project deadline is too close so that a feasible network cannot be created. A way out of this dilemma is *schedule compression*. This means that the schedule is restructured, either by speeding up activities or by performing activities in parallel that normally would be done one after the other. Schedule compression usually results in additional cost and therefore has to be considered carefully.

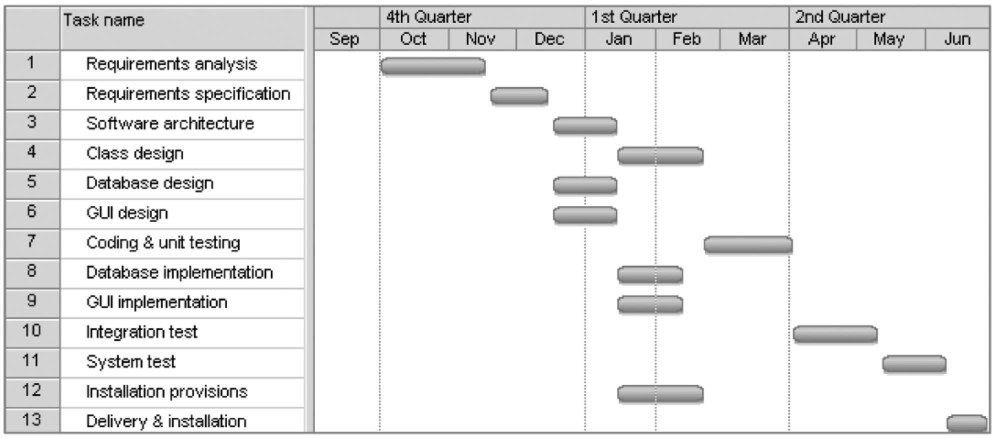
Simulation

Simulation can be used to examine schedule modifications and their consequences. *What-if simulations* help to evaluate the effects of different scenarios (e.g., what happens when the installation of the project's new IT infrastructure is delayed by one month?) and project settings. Stochastic influences can be treated with the help of *Monte-Carlo simulations.* In this type of simulation, probability distributions are used for input variables (e.g. activity durations), and possible outcomes are also presented in the form of probability distributions.

Bar charts

Schedules are usually displayed in graphical form, as bar charts or network charts. *Bar charts* exhibit activities with start and end dates and their durations on a time axis. They may also contain milestones and logical relationships between activities. Figure 8-8 shows a simple bar chart.

Figure 8-8 **Project schedule as a bar chart**



Network graphs exhibit primarily the logical dependencies between the activities in the form of activity-on-node (or activity-on-arrow) graphs. However, a network graph can contain more information such as an activity's earliest and latest start and end dates, the duration and the float. Examples of network graphs were given in figure 8-5 and 8-5.

8.2.2 Planning the Project Organization

Projects are executed within an organization (e.g. a company), and in addition, a project as such has its own internal organization. The project's organization is not independent from the surrounding organization that carries out the project. Availability of resources, the project manager's authority, budget control and many more factors depend on the organization of the company. Therefore we have to look at basic organizational structures first before discussing an ISD project's internal organization.

Organizational structure

There are three basic types of organizational structure: functional, projectized and matrix organizations. These are illustrated in the four parts of figure 8-9, adapted from the PMBOK [PMI 2004, pp. 29-31].

➡ A *functional organization* is structured according to business functions such as marketing, production, controlling etc. Staff members are grouped by specialty, i.e. people with similar skills are assigned to the same unit. Each employee has one clear superior. A dedicated project manager is optional.

A strictly functional organization

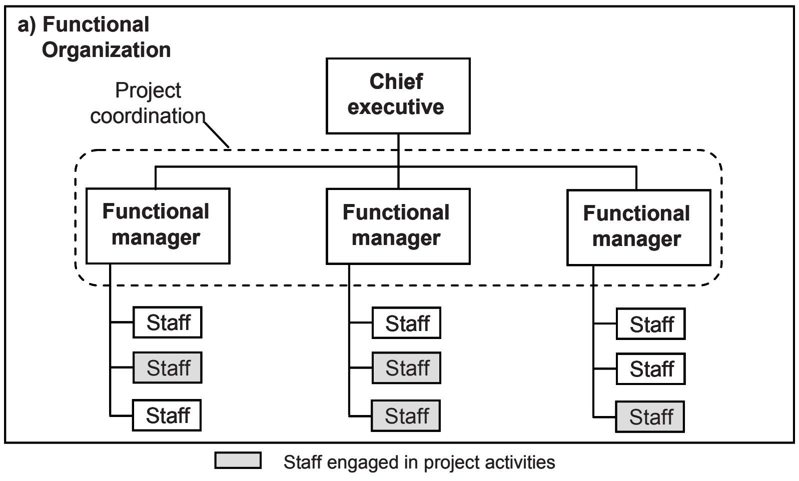
Part a) of figure 8-9 sketches a *strictly functional* organization. Note that it does not exhibit a project manager. When a project is carried out in such an organization, staff from several functional areas have to work together. Since they still report to their functional managers – both inside and outside the project – the functional managers have to collaborate in the coordination of the project.

While the advantages of a functional organization are clear responsibilities and authorities, project coordination and control are difficult and less effective. Consider, for example, a change request by an employee in the right-hand branch of the hierarchy, requiring work to be done by an employee in the left-hand branch. The issue has to be communicated to the functional manager on the right, up to the chief executive (at least if it is controversial), down to the functional manager on the left and then down to the employee who will do the work. Note that figure 8-9 shows only a two-level hierarchy. In a real organization – in a multi-level hierarchy – the path upwards and downwards the organizational tree can be long and time-consuming.

Project managers are the essential management positions

➡ A *projectized organization* (often called a *project organization*) as illustrated in figure 8-9 b) is one where the organizational structure is completely project-oriented. This could be the case when a company's business is doing projects for other companies. The essential management positions are project managers. Each project has its own staff. Employees in a project report to their project manager. Project managers have a great deal of independence and authority, including budget and resource control.

Figure 8-9 **Forms of organizational structure [PMI 2004, pp. 29-31]**



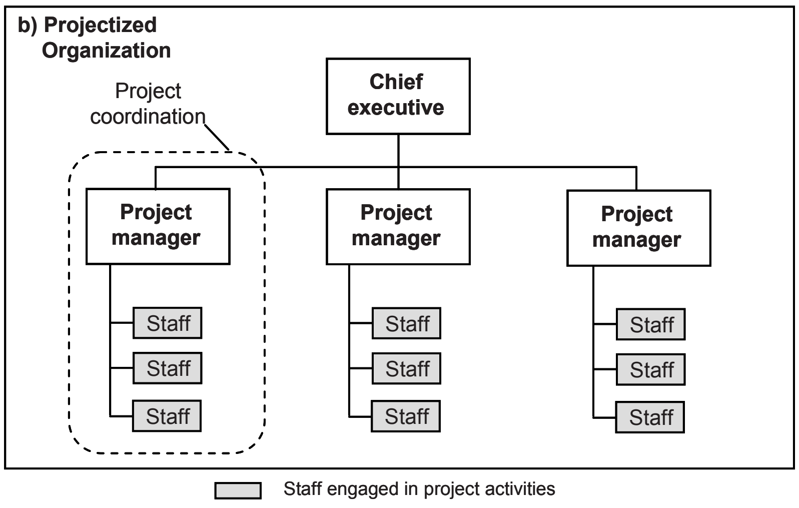
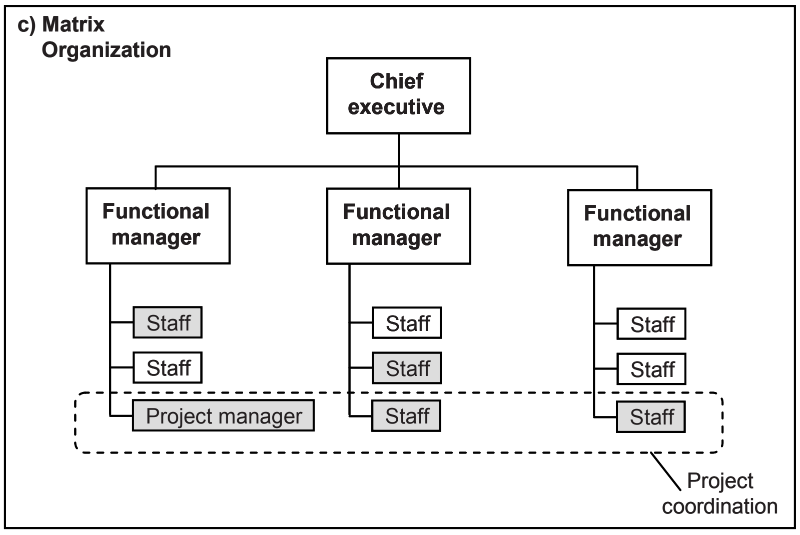
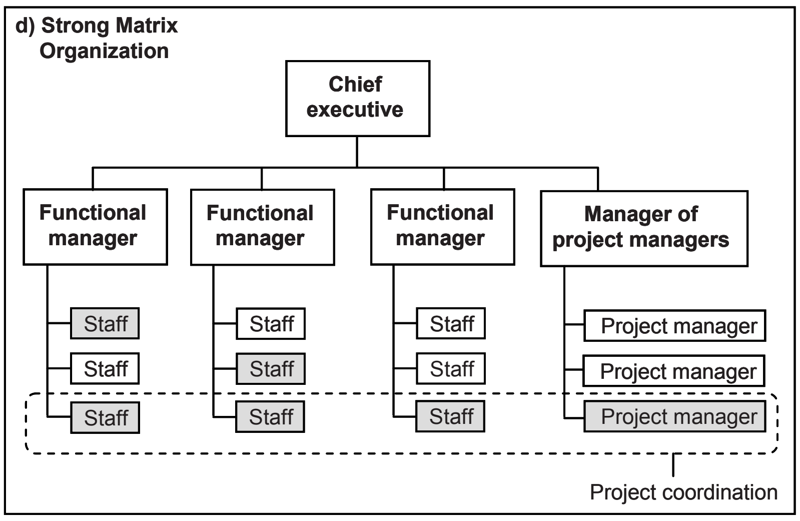


Figure 8-9 (continued)



▭ Staff engaged in project activities



▭ Staff engaged in project activities

Functional and project-oriented features

➡ *Matrix organizations* are between strictly functional and strictly projectized organizational structures, exhibiting both functional and project-oriented characteristics. Part c) of figure 8-9 sketches a matrix organization. While the basic organizational structure is functional, a cross-functional project structure is overlying the functional structure.

More authority for project managers

When a dedicated project manager is introduced, this organization is called a *balanced matrix*. Project managers have a certain degree of independence, including partial control of the budget and the resources.

➡ In a *strong matrix organization,* the role of the project managers is strengthened in that they have more authority as well as more budget and resource control. As part d) of figure 8-9 suggests, project management can be a department of its own with a manager at the top, meaning that project managers do not report to functional managers but to the manager of the project management group.

A strong matrix organization can be found in software companies that have many projects going on at the same time. Since projects vary, the organizational structure changes over time. Both matrix and projectized organizational structures have to cope with permanent change.

Figure 8-10 **Influence of organizational structure on projects [PMI 2004, p. 28]**

Organization Structure / Project Characteristics	Functional	Matrix		Projectized
		Balanced Matrix	Strong Matrix	
Project manager's authority	Little or none	Low to moderate	Moderate to high	High to almost total
Resource availability	Little or none	Low to moderate	Moderate to high	High to almost total
Who controls the project budget	Functional manager	Mixed	Project manager	Project manager
Project manager's role	Part-time	Full-time	Full-time	Full-time
Project management administrative staff	Part-time	Part-time	Full-time	Full-time

The organizational structure imposes limitations on the scope of project management. In other words: what can be decided within a project – by the project manager – depends to a great deal on the organizational structure. Figure 8-10 summarizes the influence of organizational structure on projects.

Information systems development by user organizations

Developing an information system is typically an endeavor carried out as a project. However, a number of differences from projects in other areas exist, as we pointed out in section 8.1.3.

A typical organizational structure for IS development projects within a *user organization* (i.e. a company whose core business is not software, cf. section 1.3.3) is a matrix organization.

Line-and-staff organization

Suppose the company has a basically functional organization or an organization along product lines (divisional organization). Since IT (including software development) is not a core business function or a part of a business function, it is attached as a staff department to a management instance (e.g. the CEO or a second or third-level top manager). In organizational studies, this is called a *line-and-staff organization*.

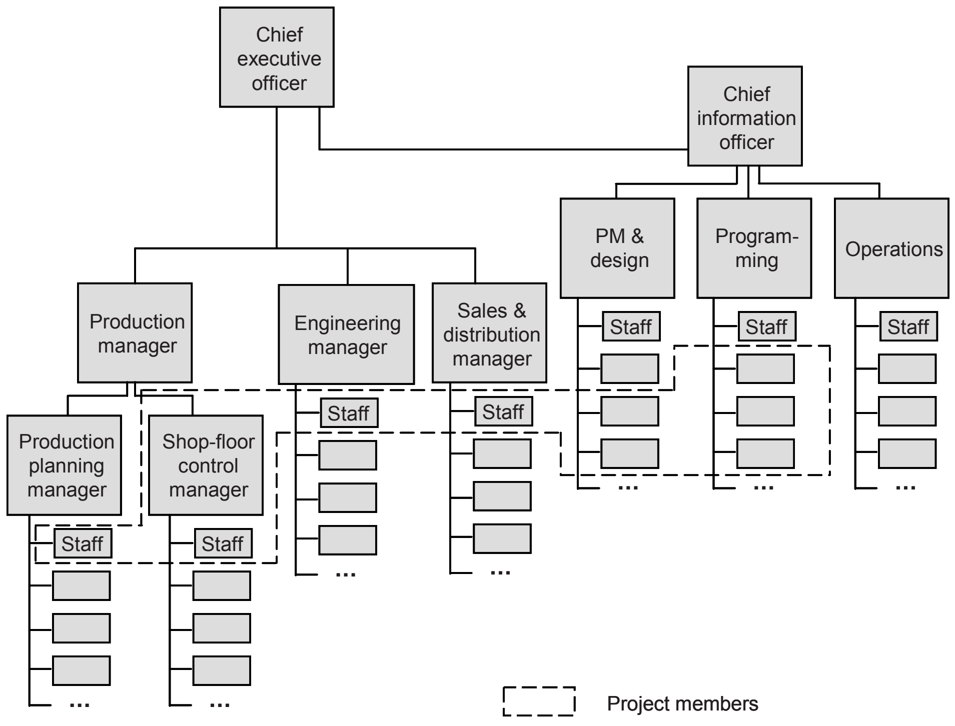
Let us assume that the manufacturing department of the company needs a more flexible shop-floor scheduling system. A project team will be set up from staff of functional areas such as production planning, shop-floor control, engineering, sales and distribution and of course from IT, as shown in figure 8-11. The business, engineering and manufacturing people remain in their functional areas. They report to their functional managers, but regarding project matters they report to the project manager as well. The project manager is likely to come from the IT department, just like the system analysts and programmers.

The project in figure 8-11 consists of ten persons, five from the IT department, two from shop-floor control, and one each from engineering, production planning and sales and distribution. Not all will work full-time for the project (e.g. the shop-floor manager), and some will need to become involved at different times (e.g. the production planner for requirements engineering, programmers for coding).

Organizational structure of a software company

Projects in organizations whose business *is* software development are different from figure 8-11 for three reasons. Firstly, software companies do not produce physical goods and thus they do not have departments such as production and engineering. Secondly, since information processing is the object of their work, there is no separate CIO and no separate IT department. Thirdly, almost all work is done in the form of projects.

Figure 8-11 **Organization of an ISD project (example)**



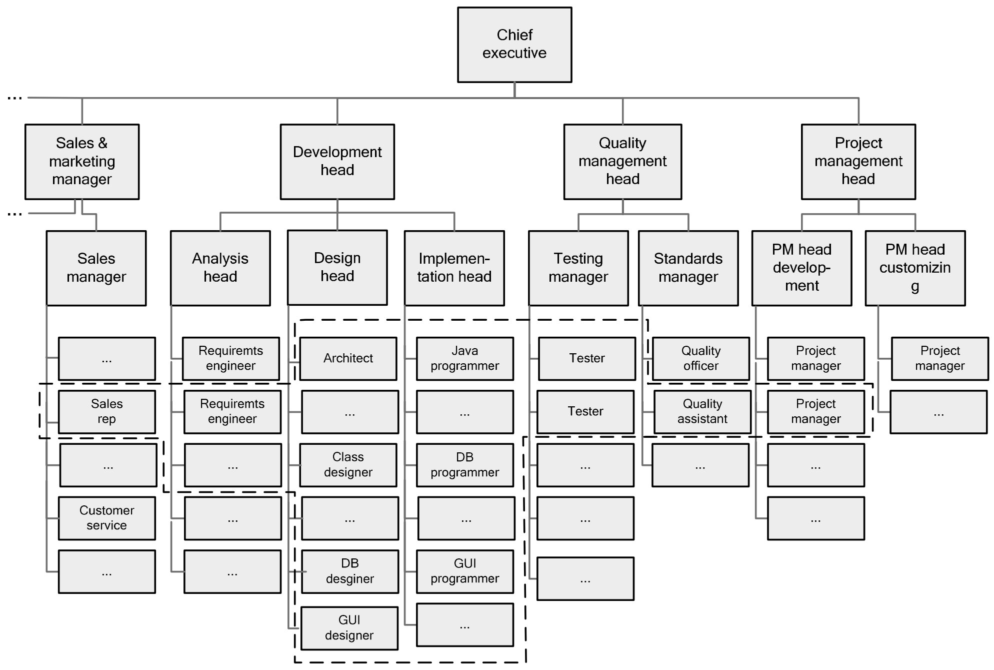
Depending on what other business the company has in addition to software development, different organizational structures are possible. For example, if IT infrastructure services (IIS) or application service providing (ASP) are part of the business, then even characteristics of a functional organization can be observed.

Strong matrix organization

Since our focus is information systems development, we will primarily consider software-development organizations and software-development departments of IT organizations. A typical organizational structure here is a *strong matrix organization*.

However, speaking of a matrix organization in a software company is different from speaking of a matrix organization in a manufacturing firm or a bank. "Matrix organization" means that functional and cross-functional features are blended. Functional areas of a software company are analysis, design, programming, testing etc., in addition to marketing, accounting and other business functions. These functional areas may be arranged in a hierarchy as shown in figure 8-12.

Figure 8-12 **Functional organization of a software company**



The figure shows a hypothetical organization structure. Since we are assuming a software company that earns its money from software development, the biggest portion of the structure is occupied by the development department. Other functional areas are sales and marketing (where sales representatives and customer support are working), quality management and project management, in addition to business functions that are present in all companies.

A hypothetical ISD project

A typical development project in this hypothetical company comprises staff from many functional areas. Assuming a strong matrix organization, the following departments will be involved:

- Project management – project manager, assistant project manager (in large projects) and administrative personnel
- Sales and marketing – e.g. the sales representative who acquired the project and serves as primary customer contact
- Analysis – requirements engineers or systems analysts performing requirements engineering
- Design – software architects developing the basic architecture of the system; class, database and GUI designers
- Implementation – programmers implementing the classes (e.g. Java programmers), DDL and DML programmers implementing the database, and GUI developers implementing the web front-ends (markup-language and script programmers)
- Testing – staff performing module, integration and system testing
- Standards – a quality officer or assistant to ensure that software-engineering standards are met.

The persons inside the dashed line in figure 8-12 together form the project team for this concrete project. In the design and implementation departments, one or more persons from each category (architect, class designer, Java programmer etc.) are involved, which is indicated by three dots interrupting the dashed line. The internal organization of such a project team is discussed in the next subsection.

Project organization within a software company

The organizational structure of a specific project is established when the project is launched. Since it is unlikely that the new project will be totally different from previous projects, templates may be available, or an earlier project organization is used as a pattern.

An organizational structure of a software development project defines the *roles* of project members and arranges the roles in a tree-like or network structure. Examples of roles are programmer, architect, domain expert, quality assistant etc. Roles are filled by persons from within the company's existing organizational units.

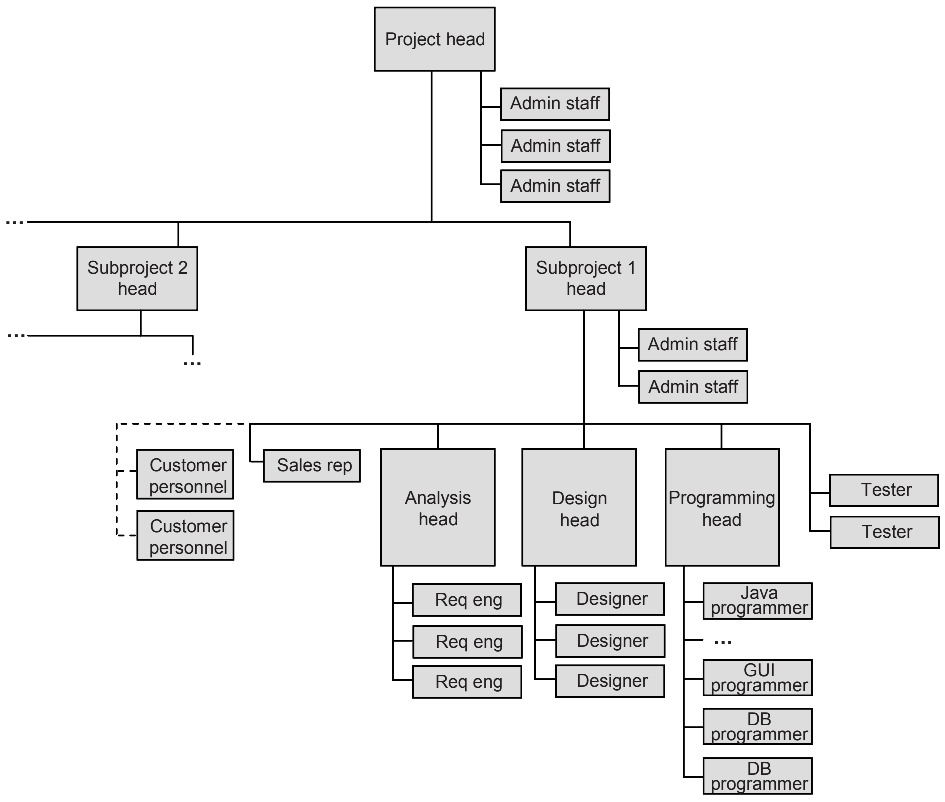
Basic project-organization types are a hierarchical organization and a team organization. Another type with both hierarchical and team characteristics is the chief programmer team.

Hierarchical project organization

In a *hierarchical project organization*, roles are arranged in a tree-like structure, with a head of the project on top and subheads with line authority at the nodes (cf. figure 8-13). The head on top is usually the project manager, but other constructions such as a technical project head plus a business head with split responsibilities are also possible. Subheads are responsible for subprojects or for certain groups of project members. Examples are a requirements manager with authority to issue instructions to requirements engineers (or system analysts) and a head of programming.

The project hierarchy can be supplemented by staff roles, in particular administrative roles supporting the project head (line-and-staff organization). In large projects, subheads may also be assisted by staff roles (e.g. a coordinator managing authorizations and software configurations).

Figure 8-13 **Hierarchical project organization**



A hypothetical sub-project

Figure 8-13 shows a line-and-staff organization of a large hypothetical project that is subdivided into subprojects. A total of 27 people are working in subproject 1:

1	subproject head
2	administrative staff
1	analysis head
3	requirements engineers
1	design head
3	designers
1	programming head
10	programmers
2	testers
1	sales representative
2	customer representatives

Customer personnel involved in the project are not under the authority of subproject head 1. Therefore a dashed line connecting these two roles is drawn in figure 8-13.

Large projects are often organized in a hierarchical way. Many large bureaucratic organizations prefer hierarchical project structures in information systems development. However, followers of non-conventional approaches (such as agile development) usually favor less formal structures as discussed in the next paragraph.

In a hierarchical structure, communication paths are formalized, going up and down the tree. If the project head in figure 8-13, for example, is exposed to an urgent customer change request regarding realization of a program function, he or she will not talk to the Java programmer directly but to the subproject 1 head. This person will talk to the programming head who will communicate with the programmer. (The reply then travels the same path up the hierarchy.)

Team organization

Small projects or subprojects are often organized in a less formal way, as a *team*. The project task is assigned to a group of people. While a project head is usually nominated by the superiors launching the project, the other roles are less clearly defined beforehand. "Everybody talks to everybody".

Suppose for example that four developers and one tester are assigned to the project. Who will do the architectural design? Who will examine and refine the requirements? Who will design classes and implement them? Rough roles and responsibilities are likely to be defined by the project manager, e.g. in such a way that a senior developer designs the software architecture, and certain parts of the total functionality are assigned to the other three developers. This means that one person performs the design, implementation, testing, documentation and perhaps even requirements analysis for one portion of the system. (Remember that XP practices such as the "planning game" and "user stories" imply that the programmer talks directly to the customer; cf. section 4.4.1).

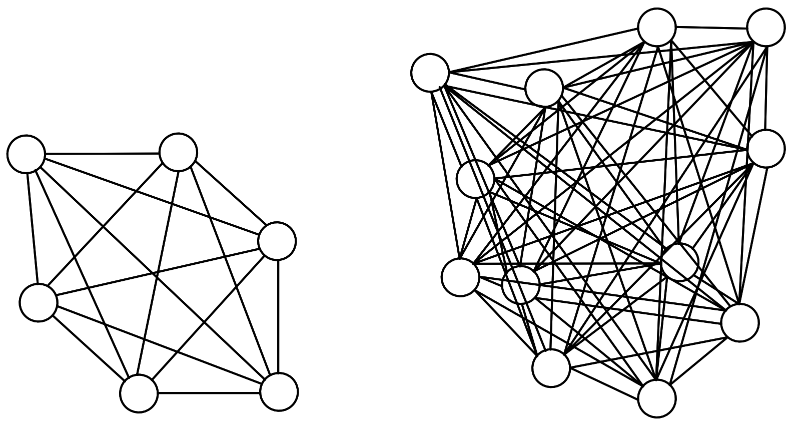
In the software-engineering community, team organization enjoys a high popularity because it exhibits democratic features. Project issues are discussed by the team and decisions are made by consensus within the team. In contrast, a hierarchical organization is characterized by "instructions" that are issued by higher-level organizational roles and executed by lower-level roles.

"Everybody talks to everybody"

A team organization can work well, provided that the team is capable and willing to collaborate in a consensual way and that the project manager has leadership capabilities such as being able to solve conflicts and motivate the team members. On the other hand, a team organization based on an "everybody talks to everybody" approach reaches its limits as the project size grows. In a large team, the communication overhead explodes, incapacitating effective team work.

For example, consider a team of 6 members (n = 6) as shown in figure 8-14 a). Each member has to talk to 5 other members, yielding n*(n-1)/2 = 15 communication paths. Suppose the team size doubles. Then each team member theoretically has to communicate with 11 people, and the number of paths increases to 66 as shown in part b).

Figure 8-14 **Communication paths in a democratic team**

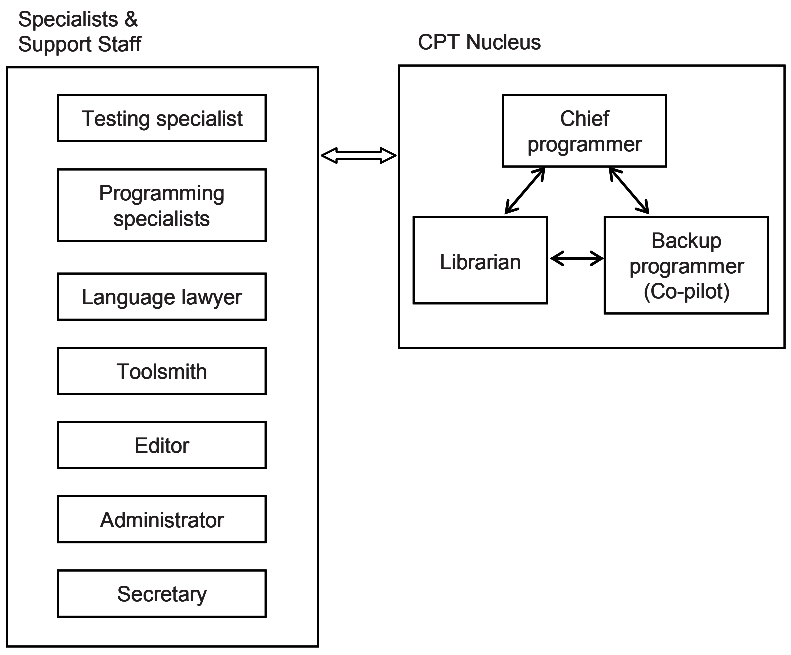


Although in practice not everybody will need to talk to everybody, the example shows that a large share of the daily working hours will be spent on communication and not on productive work, unless some hierarchical coordination mechanisms is in place.

Chief programmer team (CPT)

A *chief programmer team (CPT)* is a project organization that has been enjoying a great deal of popularity in the software-engineering literature for several decades. It is basically a hierarchical organization with most authority combined in one person (the chief programmer) and a number of roles assisting the chief programmer. This organizational structure goes back to an IBM project in the early 1970s in which an information system for the New York Times was developed [Baker 1972, Mills 1973].

Figure 8-15 **Chief programmer team organization**



The rationale for the CPT approach is based on the observation that programming abilities amongst software developers vary extremely. "The best programmers may be up to 25 times as productive as the worst programmers. It therefore makes sense to use the best people in the most effective way and provide them with as much support as possible." [Sommerville 2007, p. 605].

When the chief programmer team organization was first proposed, it was compared with a surgical team in which a chief surgeon is supported by a team of specialists whose members assist the chief rather than operate independently. As shown in figure 8-15, the permanent roles in the original CPT organization were called chief programmer, backup programmer and librarian [Baker 1972, pp. 57-58]:

– A *chief programmer* is a senior-level programmer who is responsible for the development of a software system. The chief programmer produces a critical system nucleus in full, specifies all other parts and integrates them.
– A *backup programmer* (sometimes called co-pilot) is also a senior-level programmer supporting the chief programmer. When the chief programmer is unavailable, the backup programmer can take on his or her tasks.
– The *librarian* may be a programmer, a technician or a secretary with additional technical training, responsible for clerical functions such as configuration management, tracking project progress and preparing reports.

This nucleus may be extended by including other programmers, analysts, technicians and specialists from certain fields, depending on the size and character of the system under development. Roles outside the nucleus have been proposed, named and described by many authors. As shown in figure 8-15, such roles can be [Mills 1973, pp. 58-61]:

– *Program testing specialist* – preparing and performing tests as defined by the chief programmer or the co-pilot
– *Programming specialists* – additional programmers with special skills to whom the chief programmer can delegate work
– *Language lawyer* – a programming language specialist who may create, for example, an optimized version of some module written by the chief programmer
– *Toolsmith* – a specialist for software tools and utility programs to be used by the chief programmer or others
– *Editor* – proofreading, formatting and producing a final version of the documentation created by the chief programmer and the co-pilot

– *Administrator* – bookkeeping for the project, including legal requirements such as contract reporting, patents and trade marks
– *Secretary* – providing secretarial support services

While the core roles in the nucleus are permanent roles, some of the above roles are temporary. For example, an editor will only be needed when the documentation is produced, and a specialist for bridge programs will only be needed when interfaces with an external software system have to be created.

Drawbacks

Although the chief programmer team organization has enjoyed much attention in the software-engineering literature, only few organizations actually use it. This is due to several reasons: First, individuals with such outstanding software development and management skills as required for the chief programmer are hard to find. Second, having most of the knowledge, experience and high-level skills concentrated in one person is risky. Third, since the chief programmer takes on all the responsibility and makes all the important decisions, the other project members may feel that their role is not recognized. Fourth, this organization is limited to small teams and thus to small projects.

For larger projects, variants and extensions of the CPT organization have been proposed, such as breaking down the overall system under development into smaller subsystems that can be handled by individual chief programmer teams. However, these variants and extensions are not used much. Concluding this subsection, we may regard the chief programmer team as an effective organizational structure for small projects. It that has a lot of productivity potential and at the same time bears a high risk.

8.2.3 Planning with a Project Management System (PMS)

A large number of PMS exist

Planning non-trivial projects is usually done with the help of a software tool – a *project management system (PMS).* On the market, a large number of PMS are available. For example, a list provided by the German project management society exhibits 181 systems [PM 2007]. They range from simple open-source systems for PCs all the way to heavyweight systems for mainframes with six or seven-digit license fees.

Figure 8-16 **Project management systems [Wyomissing 2008]**

Desktop Project Management Software

Software	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
GanttProject			●			●		
Microsoft Project			●	●		●		
OmniPlan			●			●		
Open Workbench			●			●		
Pertmaster					●			
PlanningForce		●	●			●		●

Enterprise Project/Portfolio Management Software

Software	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
AceProject		●	●			●	●	●
Achievo			●			●	●	
Artemis	●	●	●	●		●	●	●
InventX	●	●	●	●	●	●	●	●
Microsoft Project Portfolio Server	●	●	●	●	●	●	●	●
Planisware OPX2	●	●	●	●	●	●	●	●
Vertabase		●	●	●	●	●	●	●

Feature Definitions

(1)	Demand	Demand management features for requesting work, collecting requirements, etc.
(2)	Portfolio	Managing a portfolio of projects, balancing resources across a portfolio of projects, etc.
(3)	Schedule	Creating and tracking on a schedule with dependencies and milestones
(4)	Budget	Defining and tracking on project budget performance
(5)	Risk	Features for defining and mitigating risks and tracking issues
(6)	Resource	Tools for defining resources, allocating to projects, and analyzing utilization
(7)	Time	Features for timesheet entry, charging to project tasks, and reporting
(8)	Performance	Project performance metrics and analytical tools

Some well-known project management systems are listed in figure 8-16. Two simplified categories are distinguished in this figure: desktop systems and enterprise systems. The first category comprises mostly systems for the management of single projects whereas the second category addresses enterprise-wide project management (multi-project management, cf. section 8.4).

The entries in the criteria columns of figure 8-16 indicate that desktop PMS basically support project scheduling and resource management. Full-fledged enterprise PMS provide additional features such as budget and risk management, timesheets, performance measurement and features for multi-project management (demand and portfolio management).

Microsoft Project

A widely used project management system is *MS Project*. Although it falls in the category "desktop systems", it is a professional PMS providing many features for project planning and project control. MS Project goes back to the late 1980s when the first version was developed. Being part of the MS Office suite, its current official name is Microsoft Office Project 2007.

"Microsoft Office Enterprise Project Management (EPM) Solution"

As the second table in figure 8-16 shows, server-based versions of MS Project also exists. MS Office Project 2007 is actually a product family. As of 2008, Microsoft offers the following products from this family under the name "Microsoft Office Enterprise Project Management (EPM) Solution" [Microsoft 2008]:

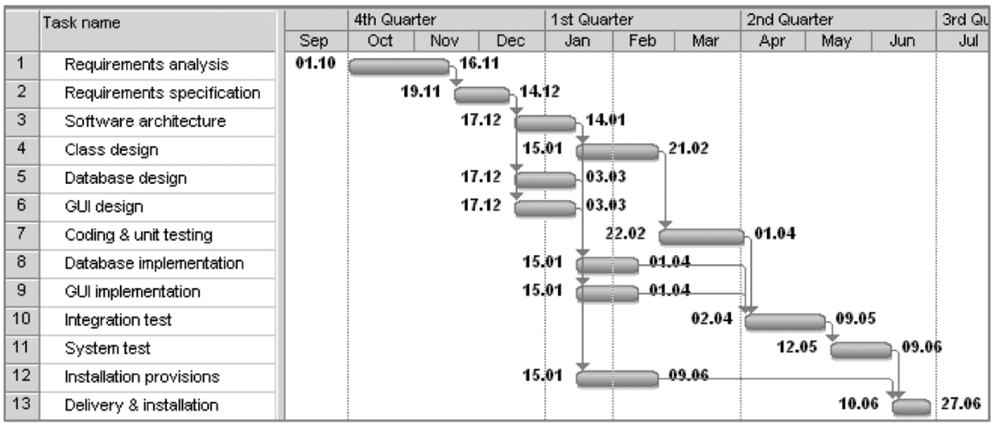
– MS Office Project Professional 2007 – providing desktop functionality plus EPM capabilities when connected to Microsoft Office Project Server 2007
– MS Office Project Server 2007 – allowing organizations to store project and resource information centrally
– MS Office Project Portfolio Server 2007 – supporting project portfolio management (multi-project management)

Main presentation tool: bar chart

The main presentation tool provided by systems such as MS Project is a *bar chart*. This bar chart can be enriched with many types of information including an activity's start and end dates (earliest or latest dates); cost, people and other resources assigned to the activity; duration, successors and/or predecessors of the activity in the work breakdown structure (WBS) etc.

The bar chart in figure 8-8 was created with MS Project. Another example of a bar chart displaying not only activities but also the network structure is shown in figure 8-17. The dates written next to the bars are an activity's earliest start date and latest end date.

Figure 8-17 **Bar chart with logical dependencies and dates**

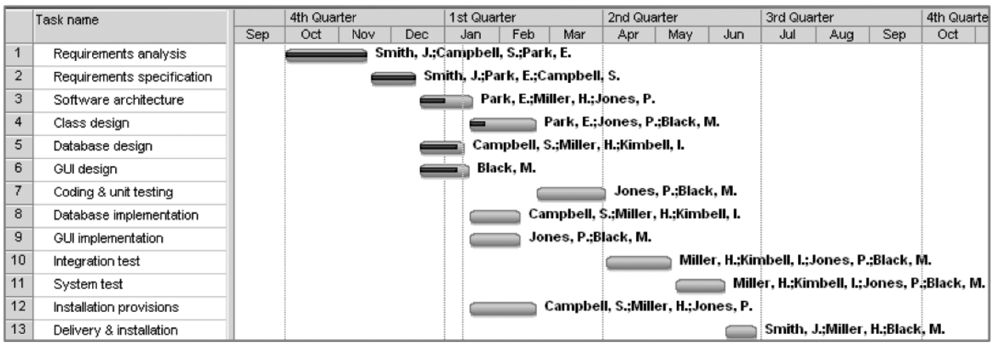


	Task name	Earliest start	Latest end
1	Requirements analysis	01.10	16.11
2	Requirements specification	19.11	14.12
3	Software architecture	17.12	14.01
4	Class design	15.01	21.02
5	Database design	17.12	03.03
6	GUI design	17.12	03.03
7	Coding & unit testing	22.02	01.04
8	Database implementation	15.01	01.04
9	GUI implementation	15.01	01.04
10	Integration test	02.04	09.05
11	System test	12.05	09.06
12	Installation provisions	15.01	09.06
13	Delivery & installation	10.06	27.06

Legend: Left – earliest start date, right – latest end date

For example, the "GUI implementation" activity cannot start before January 15th, and it must end not later than April 1st. MS project can generate and display these dates because the activity sequences, durations and dependencies were defined when the activities were created.

Figure 8-18 **Assigning resources to activities**



	Task name	Resources
1	Requirements analysis	Smith, J.;Campbell, S.;Park, E.
2	Requirements specification	Smith, J.;Park, E.;Campbell, S.
3	Software architecture	Park, E.;Miller, H.;Jones, P.
4	Class design	Park, E.;Jones, P.;Black, M.
5	Database design	Campbell, S.;Miller, H.;Kimbell, I.
6	GUI design	Black, M.
7	Coding & unit testing	Jones, P.;Black, M.
8	Database implementation	Campbell, S.;Miller, H.;Kimbell, I.
9	GUI implementation	Jones, P.;Black, M.
10	Integration test	Miller, H.;Kimbell, I.;Jones, P.;Black, M.
11	System test	Miller, H.;Kimbell, I.;Jones, P.;Black, M.
12	Installation provisions	Campbell, S.;Miller, H.;Jones, P.
13	Delivery & installation	Smith, J.;Miller, H.;Black, M.

Resources (including people) can also be stored and assigned to those activities for which they are needed. Figure 8-18 illustrates the assignment of project workers to activities. It should be noted, however, that MS Project supports only the creation of "flat" resources and the assignment of these resources to activities. It does not support the modeling of an organizational structure such as a hierarchy as shown in figure 8-13.

8.3 Project Execution and Control

Coordinating people, resources and activities

To accomplish the project's requirements, the work specified in the planning phase must be carried out. This involves the coordination of people, resources and activities within the scope of the project. The activities of the project must be initiated, performed, supervised and controlled. The project management will monitor the timely creation of the deliverables specified in the project plan, consider change requests, initiate corrective actions, and update the project plan.

In principle, all partial plans drafted for the above mentioned project management areas – integration, scope, cost, quality, human-resources, communications, risk and procurement management (cf. section 8.2.1) – will be implemented in the execution phase. Monitoring and controlling the project's progress as well as re-planning and reacting to change are the major challenges for the project management.

8.3.1 Monitoring and Controlling Project Work

Monitoring, controlling, reporting

The main objective of monitoring and controlling is to compare actual project performance against the project plan (including all partial plans) and to take corrective or preventive actions if needed. Project *monitoring* means collecting, measuring and disseminating performance information [PMI 2004, p. 94]. *Controlling* compares the measurements against the project baseline and takes any necessary remedial action.

Monitoring and control are usually accompanied by *reporting*. This means creating reports for the management about the status of work on the project, including forecasts of developments up until the end of the project [IPMA 2006, p. 72].

In more detail, monitoring and controlling the project work include the following tasks [PMI 2004, p. 94]:

– Comparing actual project performance against the agreed project plan.
– Assessing performance to determine whether any corrective or preventive actions are indicated, and initiating these actions if necessary.
– Analyzing, tracking and monitoring project risks to make sure the risks are identified, their status is reported, and that appropriate risk response plans are being executed.
– Maintaining an accurate, timely information base concerning the project's results and associated documentation.
– Providing information to support status reporting, progress measurement and forecasting.
– Providing forecasts to update current cost and current schedule information.
– Monitoring implementation of approved changes when and as they occur.

To satisfy these requirements, monitoring and controlling tasks have to be carried out within the project management areas, in particular in scope, schedule, cost, quality and risk management. The PMBOK, for example, describes a set of monitoring and controlling tasks in these areas in detail [PMI 2004, sections 5.5, 6.6, 7.3, 8.3 and 11.6]. In brief, they can be summarized as follows:

Scope control

Avoiding scope creep

Scope control is concerned with scope changes and the impacts of these changes. Scope control tries to avoid the so-called *scope creep* (uncontrolled growths and changes of requirements, cf. section 8.1.3), managing changes through a defined process of change requests, change approvals and recommended actions to carry out changes. As a result, the work breakdown structure (WBS) and the scope statement may be updated.

Schedule control

Schedule control is a task which determines the current status of the project schedule, to examine if and why the schedule has changed, and to manage actual changes as they occur [PMI 2004, p. 152]. For schedule control, the start and end dates of the activities as shown in figure 8-15 and the project milestones are used. Schedule control includes progress reporting and a defined process by which the schedule can be changed (request, approval, authorization etc.). An important part is to decide if a detected schedule variation requires corrective action. For example, a delay on a scheduled activity which is not on the critical path may have little effect on the overall project schedule and can be tolerated.

Comparison bar charts

Schedule analysis is facilitated by comparison bar charts plotting the approved project schedule against the actual project schedule. This is a visual help showing where the schedule has progressed as planned and where not.

Outputs of schedule control

Outputs of schedule control include updates of the project schedule, in particular updated network diagrams and bar charts with effective new start and end dates. Any schedule change can have an impact on other project management areas. For example, if an activity is rescheduled, as a consequence, resources needed for this activity may also have to be reallocated, according to the new start and end dates.

As another output of schedule control, corrective actions may be defined in order to bring the expected future schedule back in line with the approved schedule. This includes actions to ensure completion of schedule activities on time or with minimum delays.

Cost control

Cost control deals with the cost performance of the project. The time-phased budget of the project is monitored, actual expenditures and variances against the approved budget are determined, and requested changes to the budget are agreed upon. Cost control includes the measurement of project performance and forecasting future project developments, in particular their cost implications.

Taking into account that cost overruns are rather common in practice, a crucial task of cost control is to ensure that potential overruns do not exceed the authorized budget for the period or entire project. If an overrun cannot be avoided, then appropriate measures have to be taken to keep the overrun within acceptable limits. Scope or schedule change requests may also induce additional costs. If they are approved, the budget has to be adapted.

Earned value technique (EVT)

A controlling technique for performance measurement is the *earned value technique (EVT)*. This technique "... compares the cumulative value of the budgeted cost of work performed (earned) at the original allocated budget amount to both the budgeted cost of work scheduled (planned) and to the actual cost of work performed (actual)" [PMI 2004, p. 172].

Figure 8-19 **Earned value technique example [PMI 2004, p. 174]**

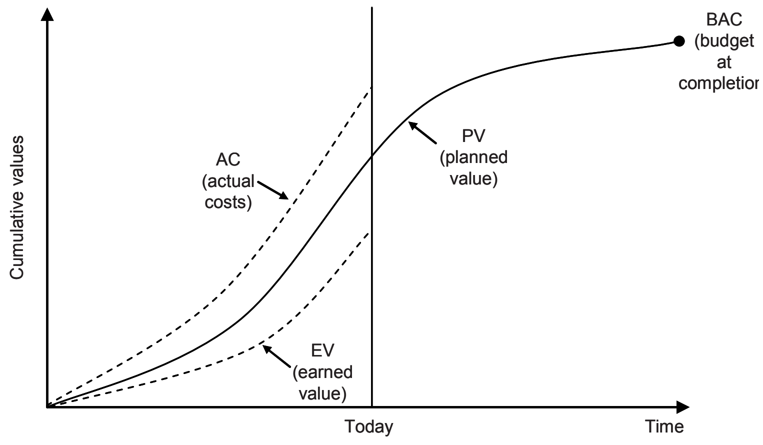


Figure 8-19 illustrates this concept for a work component that is over budget and behind the project plan. The curves and points shown in the figure are:

– *PV (planned value)* – budgeted cost for an activity (or an aggregated component of the work breakdown structure), cumulated up to a given point in time
– *EV (earned value)* – budgeted amount for the work actually completed on this activity (or WBS component) during a given time period
– *AC (actual cost)* – total cost incurred in accomplishing work on the schedule activity (or WBS component) during a given time period

– *BAC (budget at completion)* – the total budget for the project, equal to the planned value (PV) at the point in time when the project is scheduled to end

At the point in time when the measurement was done, the work component in the example of figure 8-17 had a lower earned value than the planned value. This means that not as much work has been completed as was assumed in project planning.

Forecasts

Cost and schedule variances as well as performance indices can be computed from the planned, earned and actual values. These variances and indices are used to forecast the remaining work on the activity (or WBS component), in particular to compute the:

– ETC (estimate to complete) – an estimate of the cost needed to complete the remaining work. For example, ETC = BAC - EV, possibly weighted by a cost performance index.
– EAC (estimate at completion) – the most likely total cost based on project performance, risk quantification, remaining budget and/or actual costs. For example, EAC = AC + ETC, or EAC = AC + BAC - EV, possibly weighted by a cost performance index.

The exact ways to compute the estimates depend on an assessment of the past variances and the original assumptions for cost estimation as well as on expectations about how likely it is that similar variances will occur in the future. Readers interested in details of these forecasts and the earned value technique are advised to consult the literature [e.g. Fleming 2006].

Quality control

Project quality has two aspects: product quality and process quality. The causes of unsatisfactory results have to be identified and eliminated. In information systems development, quality control includes monitoring of both the quality of the information system under development and the quality of the development process.

Product quality

An information system's *product quality* is usually addressed under the topics software quality management, validation, verification or testing. Concrete measures to ensure and improve an information system's quality were discussed in section 6.3, for example testing strategies and test-driven development. It is the project manager's responsibility to control and ensure that quality assurance measures – in particular creating a good design, evaluating development documents regarding validation and verification of results, and doing thorough testing – are adequately performed on the system under development.

Process quality

Process quality is influenced by factors such as: is the process model suited for the particular development effort and/or how well was it tailored to the needs of the current project? Are the test documents (e.g. test specifications, test-case descriptions) appropriate? Is the process well documented and followed by the development team?

CMMI (Capability maturity model integration)

Monitoring and improving processes is a continuous task. This can be based, for example, on the CMMI (capability maturity model integration). CMMI is a process improvement approach that describes effective processes and helps organizations to establish such processes [SEI 2007].

Risk control

Monitoring and controlling risks requires that potential risks were identified and possible responses were defined before the information system construction began. The project management has to keep track of known risks and identify, analyze and plan for newly arising risks. Risk monitoring and control can lead to a re-assessment of known risks. The project management is also responsible for the execution of risk responses and the evaluation of their effectiveness. They may recommend project changes as well as corrective and preventive actions (including contingency plans) to bring the project into compliance with the project goals [PMI 2004, pp. 266-268].

8.3.2 PMS Support for Project Controlling

Monitoring and controlling a project effectively is much easier with the help of a project management system. Provided that all planned activities, dates, dependencies, resources, costs etc. were entered during the planning phase of the project, it is fairly easy to have ongoing checks performed automatically whether or not the planned data have been put into effect and/or deviations have occurred.

Based on the differences between planned and actual values, the schedule can be adapted, start and end dates can be recalculated, and cost and time estimates can be adjusted automatically. This is possible if the actual state of the project is continously updated, i.e. if all starts and ends of activities and all resources utilized are booked in the PMS.

MS Project

Some project-controlling support as provided by MS Project is illustrated in the figures 8-20 to 8-23.

Cost controlling

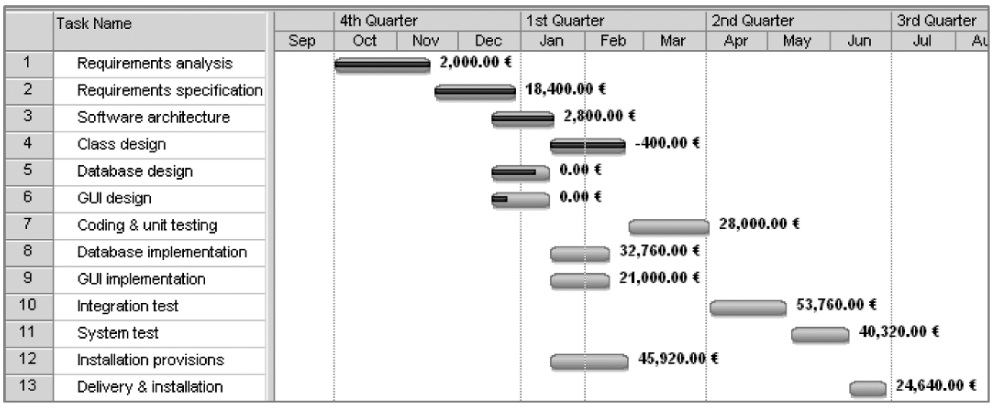
Figure 8-20 is used for *cost controlling*. The solid bars inside some of the activity bars indicate activities that have been started but are not completed yet. The inner bars visualize how much of an activity is completed (%) at the time the chart is generated.

For the *completed activities* in figure 8-20, cost variances were computed and written to the right of the activity bars. For example, requirements analysis cost 2,000 € more than planned, and class design was completed at 400 € less than expected.

Current activities are "database design" and "GUI design". Since these activities are not yet completed, their cost variances are 0 €. (Alternatively, the project manager could have displayed the actual cost spent so far or the remaining cost for these activities, if he or she was interested in more details.)

For *activities not yet started* (i.e. all activities from "coding & unit testing" to "delivery & installation"), the planned costs are displayed. "System test", for example, was planned to be completed at 40,320 €.

Figure 8-20 **Cost controlling with MS Project**



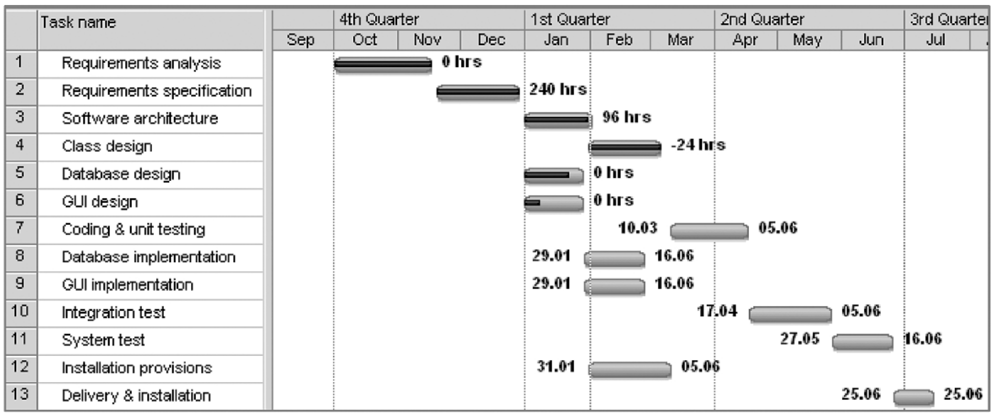
	Task Name	
1	Requirements analysis	2,000.00 €
2	Requirements specification	18,400.00 €
3	Software architecture	2,800.00 €
4	Class design	-400.00 €
5	Database design	0.00 €
6	GUI design	0.00 €
7	Coding & unit testing	28,000.00 €
8	Database implementation	32,760.00 €
9	GUI implementation	21,000.00 €
10	Integration test	53,760.00 €
11	System test	40,320.00 €
12	Installation provisions	45,920.00 €
13	Delivery & installation	24,640.00 €

Legend: Tasks 1-6 cost variance, 7-13 planned cost (baseline cost)

Schedule controlling

Schedule controlling is illustrated in figure 8-21. For completed and ongoing activities, the work variances (i.e. difference between planned and actual time spent) are shown. For example, the requirements specification took 240 hours more than planned, while requirements analysis was completed as expected (variance 0 hours) and class design took less time than planned (variance -24 hours).

Figure 8-21 **Work variances and upcoming start and end dates**



	Task name
1	Requirements analysis
2	Requirements specification
3	Software architecture
4	Class design
5	Database design
6	GUI design
7	Coding & unit testing
8	Database implementation
9	GUI implementation
10	Integration test
11	System test
12	Installation provisions
13	Delivery & installation

Legend: Tasks 1 - 6 work variance, 7 - 13 earliest start and latest end dates

Activities not started yet are shown with their earliest and their latest possible start dates. For example, database implementation could start on January 29th, and it must not start later than May 30th, otherwise the project completion date would be exceeded.

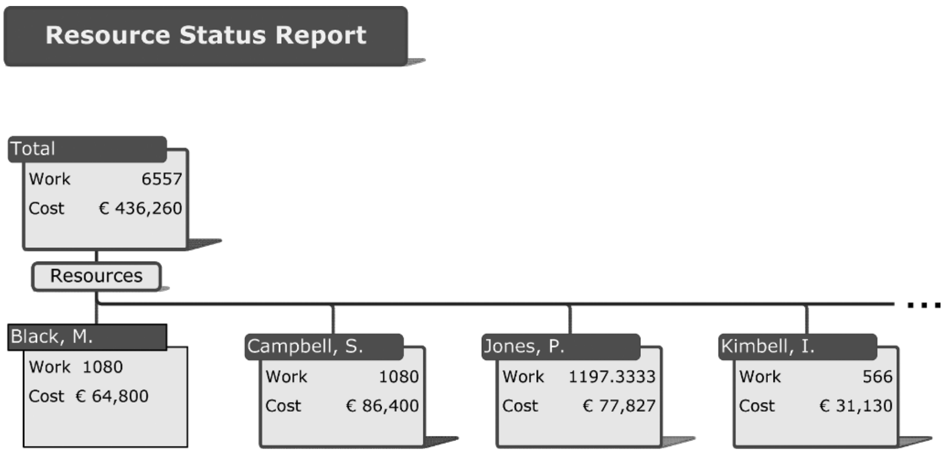
Also interesting for the project manager and the stakesholders who are financing the project is how the resources are utilized. Since all necessary data are stored and updated in the PMS – and provided that they are regularly updated – the PMS can generate reports such as the status of resource utilization.

Utilization of resources

Figure 8-22 shows such an evaluation. It was created by MS Project, based on the project data in its database. MS Project automatically exported the data to MS Visio and made Visio create a chart. This chart is a one-level tree extending as far to the right as there are resources. Since we cannot display the whole tree within the print area of a book, it is cut off after four project workers on the right.

The resource status report shows that at the time it was created, a total of 6,557 work hours and 436,260 € were already spent on the project staff, divided up between the individuals as displayed on the lower nodes of the tree. For example, Paul Jones worked 1,197 hours on the project, charged to the budget at 77,827 €.

Figure 8-22 **Resource status report**



8.4 Multi-project Management

In real-world business settings, organizations usually have not only one project but many projects going on at the same time. These projects may be independent of each other or related with one another. Multi-project management is concerned with environments in which multiple projects have to be selected and managed.

8.4.1 Levels and Tasks of Multi-project Management

Strategic level

On the *strategic level,* it is a senior management task to decide which projects to do and which budget and resources to allocate for each project, taking into account that projects may have higher or lower priorities, promise different benefits, and compete for the same resources. Portfolio analysis as described in section 2.2 is a method to evaluate and select candidate projects.

Operational level

On the *operational level,* parallel and/or sequential projects have to be coordinated with regard to overlapping staff, cash-flow and resource requirements. If results from one project are needed for another project, additional dependencies have to be considered.

Managing multiple projects is a much more complex task than managing a single project. Topics such as integration, scope, time, human resource, cost, quality, communication, risk and procurement management have to be addressed for all the projects. What makes the management even more difficult is that the projects can have different objectives and most likely they are in different stages. For example, some may be just in the requirements analysis stage while others are in the design stage and yet others in one of the testing stages.

Nevertheless, the senior management expects every project to be successful within the allocated times and budgets. They expect transparency regarding schedule and cost of each project; coordination of work across the projects; allocation of resources to projects according to priorities; and resolution of conflicting requirements between the projects.

Terms related with multi-project management are project portfolio management and program management. Some authors use these three terms interchangeably, while others summarize the latter two under the first one. We will follow this second way and define the terms as follows:

Project portfolio management

– *Project portfolio management (PPM)* is concerned with actively identifying, evaluating and prioritizing projects, resources and budgets within an organization. PPM helps an organization evaluate and prioritize each project according to certain criteria, such as strategic value, impact on resources, cost etc. [Greer 2006].

Program management

– *Program management* is concerned with managing multiple interdependent projects in order to meet given business goals. The focus of program management is on coordinating and prioritizing resources across projects to ensure that resource allocation is managed from an enterprise-wide perspective.

Multi-project management

– *Multi-project management* is the management of an organization's projects – both ongoing projects and candidate projects – through organization-wide processes ensuring that projects are selected, prioritized and equipped with resources in such a way that the business goals are met. Multi-project management comprises program management and project portfolio management.

Regarding the topic of this book, relevant multi-project management issues are primarily those on the operative level (program management). Important tasks include cross-project planning and controlling of shared resources, cross-project reporting, and common standards for quality management and project assessment.

8.4.2 PMS Support for Multi-project Management

Enterprise-level PMS

Enterprise-level project management systems such as the ones shown in figure 8-16 usually include features for portfolio and program management. As an example, the main features of the *InventX SP^{2}M* ("strategic project portfolio management") system are listed below. As the name suggests, the focus of this PMS is portfolio management and setting up projects. InventX supports the following areas [Cranes 2006]:

– *Portfolio management* – creating and maintaining a complete view of the scope, resources, schedules and budget performance of all or a subset of the projects, and consolidating information from multiple projects.

– *Strategic planning* – developing and maintaining a dynamic strategic plan of projects. Setting up a strategic plan includes capturing vision, mission, SWOT analysis, key strategies etc.

– *Project planning* – initiating the project and identifying starting activities that lead to the launching of a new project.

- *Requirements management* – capturing, managing and communicating the product and project requirements and specifications.
- *Resource management* – capturing resource information such as type, skills, training requirements etc. and assigning resources to project tasks; monitoring resources and making decisions regarding resource allocation across multiple projects.
- *Time & expense management* – time accounting and expense management including time sheet and expense reporting for both project and non-project activities.
- *Reports and analytics* – providing executives with a digital dashboard into the portfolio project system; making real-time project status information available.
- *Task management* – displaying project tasks assigned to individual team members, helping the team members to review and update the status of their tasks, including approval and recommendations by the project manager.

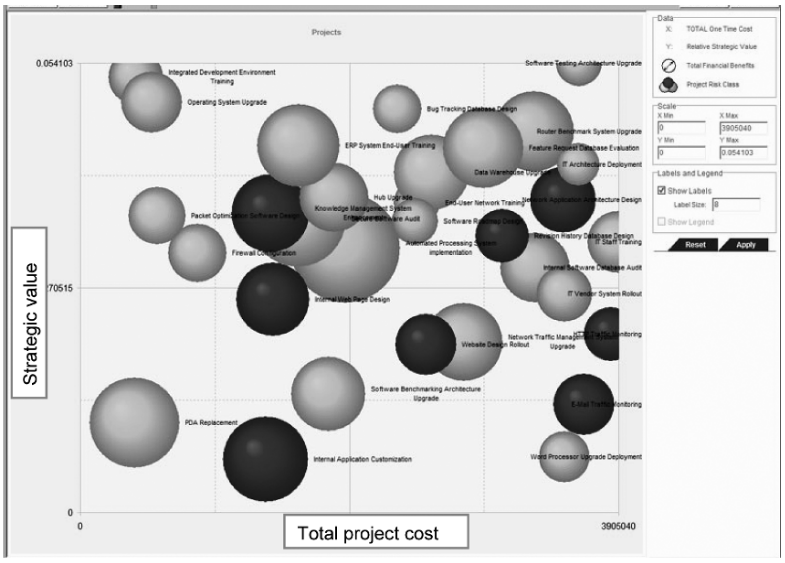
Another example of multi-project management support is MS Project ("Microsoft Office Project 2007"). This product family comprises two servers with multi-project management capabilities:

MS portfolio server

- The *portfolio server* ("MS Office Project Portfolio Server 2007") provides methods to manage all project and program portfolios with the help of workflows that subject each project to the appropriate governance controls throughout its life cycle [Microsoft 2008a]. It supports the collection of data and metrics for each project and program, storing these data and metrics in a common repository. The performance of each project or program can be measured and tracked throughout its life cycle. Furthermore, algorithms to select the optimal portfolio under varying constraints, such as costs and full-time equivalents (FTE), are provided.

  Another important feature is the automatic derivation of prioritization scores such as strategic value, financial value, risk, architectural fit and operational performance to assess projects and programs. Figure 8-23 shows an evaluation of a number of projects, generated from such scores. The projects are plotted in a two-dimensional space of total project cost (x-axis) and relative strategic value (y-axis). The bubbles represent projects, with the size of a bubble expressing the total financial benefits from a project. The colors red, green and yellow (shades of gray in the figure) stand for risk classes.

Figure 8-23 **Bubble chart to evaluate projects [Microsoft 2008a]**



Data
X: TOTAL One Time Cost
Y: Relative Strategic Value
Total Financial Benefits
Project Risk Class

Scale
X Min: 0 X Max: 3905040
Y Min: 0 Y Max: 0.054103

Labels and Legend
[✓] Show Labels
Label Size: 8
☐ Show Legend

© Microsoft Corp.

MS project server

– The *project server* ("MS Office Project Server 2007") provides central storage of all project and resource-related information. This includes high-level resource allocation for proposed projects before they are approved. The server supports the management of programs with multiple (sub-) projects and their cross-project dependencies in a coordinated fashion. Deliverables are used to track and manage these dependencies [Microsoft 2008b].

In information systems development, project and program managers need a clear picture of the utilization of resources across projects, the project budgets and costs, and the evolution of requirements. A multi-project management system can effectively assist the managers in obtaining this picture. It helps them keep track of the progress and the performance of all ongoing projects. Through diagrams visualizing aggregated information they get a quick overview of the status and how the projects are doing.

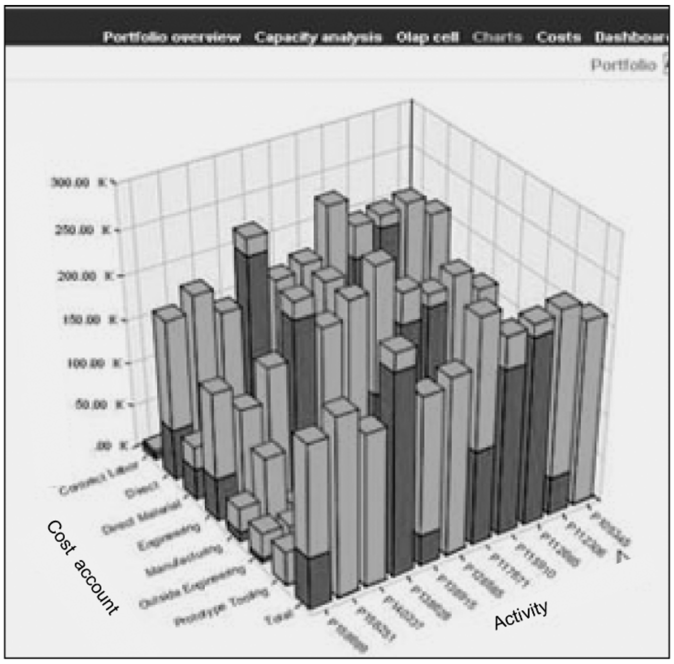
An example of an overview chart generated with the *Planisware OPX2* PMS is shown in figure 8-24 [Planisware 2007].

CPMS cost controlling support

This chart assists managers in cost controlling. It displays a number of projects, extending to the right, and the cost accounts project activities are charged to, extending towards the back. The stacked bars in the diagram represent actual cost (dark gray) and ETC (estimate to complete, light gray).

Other views, angles and chart types can be used to exhibit more information. In this way, program or project managers can see immediately how much each project has already spent and how much more is expected to be spent.

Figure 8-24 **Multi-project cost controlling support [Planisware 2007]**



© Planisware

8.5 Managing Globally Distributed Projects

Developing information systems is increasingly characterized by global work distribution. Many development teams are composed of specialists who are located in different geographical regions. It has become normal that individuals, teams, organizational units and/or organizations in different parts of the world collaborate on common projects.

8.5.1 Managing Global vs. Managing Local Projects

In large projects, distributing the work to different organizations (or organizational units) has long been practiced as a common approach. However, these organizations were usually located nearby, at least in the same country. Due to the benefits of offshoring (cf. section 2.3.3), the distribution of work is now happening on a global scale. People, teams, departments and organizations collaborate on projects even when they are thousands of kilometers apart.

GDW, GSD, GDSD

Globally distributed work (GDW) has become a common characteristic of the software industry. With regard to software engineering, a sub-discipline named GSD (global software development) or GDSD (globally distributed software development) has emerged, and international conferences and workshops are addressing this problem domain.

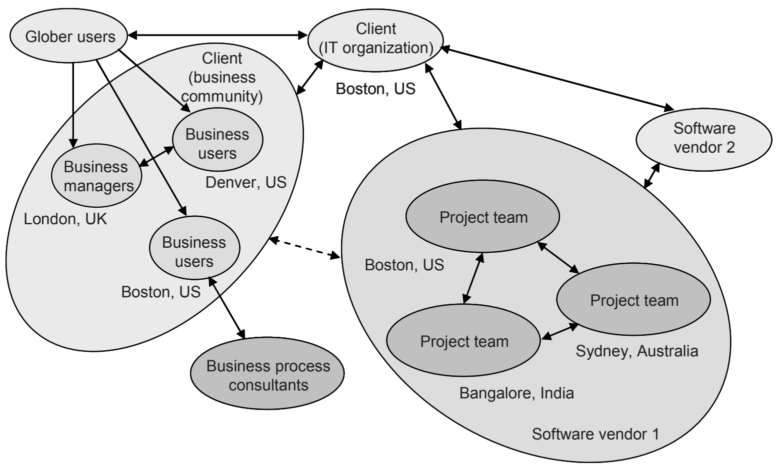
The major tasks and management areas in global software development projects are the same as in collocated projects, but there are additional issues to be solved which make the management more difficult. In particular, long distances, the cultural gap, different time zones, languages and working habits pose challenging problems to the project management.

Distributed stakeholders

The sheer fact that the stakeholders are physically separated by long distances creates many problems. As an example, consider a stakeholder structure as shown in figure 8-25. The user organization's management is in London, their IT group who runs the project is in Boston, and most of the future users are in Denver and Boston.

The software organization, headquartered in Bangalore, has project teams working in Boston, Sydney and Bangalore. They subcontracted another software firm who also needs to communicate with the client's IT organization. Although the "official" communication between the customer and the software vendor goes through the customer's IT organzation, there is indirect (and informal) communication between the two (shown as a dashed line).

Figure 8-25 **Stakeholders in a global project [Bhat 2006, p. 39]**



It is easy to imagine that communication in such a project is much more difficult than in a collocated project. It costs more money (e.g. travel expenses) and is prone to misunderstandings. Many things take longer, because requests, responses and documents have to be exchanged across time zones, cultural spheres and language barriers. For example, if partners in countries with different languages are involved, time for translating documents such as requirements specifications, designs, test plans etc. has to be taken into account.

It is a challenge for the management of a global project to define and implement solutions to overcome these problems.

GDSD issues for project management

Project management areas for global projects are basically the same as described for local projects in section 8.1.2. However, special attention has to be given to factors resulting from global work distribution. Following the PBMOK's categories, this is the case in the following areas:

– *Scope management:* According to the PMBOK, the work breakdown structure (WBS) is created in this management area. In a global project, the project manager will be thinking about work packages that can be outsourced to the offshore partner when the WBS is created. This means, he or she will define work packages in such a way that they can easily be outsourced later.

– *Time management:* Activities must be well-defined, in particular activities that will be outsourced, to avoid misunderstandings later in the project. A clear understanding of the activities by both parties is needed. The offshore partner should sign off the work-package and activity definitions.

  Buffers make sense

  Activity sequencing should allow for buffers for activities outsourced to the offshore company and for additional buffers to manage integration of results across teams. Activities on the critical path are problematic candidates for outsourcing, since the risk of exceeding end dates is higher than for local activitites.

  Time and effort estimation should be performed by both parties, because the offshore partner has a better understanding of offshore factors influencing the time and effort.

  Although much of the collaboration and communication is facilitated by electronic means nowadays, in-person meetings are still required in many cases. This means that time for obtaining visa or working permits have to be built into the time plan.

– *Cost management:* If the onshore organization has their own software development staff, "make or buy" decisions may be made for individual activities (or work packages). This means that the project manager has to analyze the costs and benefits of offshoring vs. completing the work inhouse.

– *Quality management:* Additional quality measures may be introduced, in particular when the offshore partner does not have a certified quality standard. While Indian software companies are often on a relatively high CMMI level, offshore partners in other countries may lack this certification. This leaves it to the customer to impose quality standards on the offshore partner. For example, the customer would have to verify the partner's testing procedures.

Bridging cultural gaps

– *Human resources management:* In many cases the customer wishes to control the skills and experience of the offshore organization's staff and initiate training to make sure that the necessary skills are available. Special emphasis has to be placed on bridging cultural gaps between onshore and offshore project team members. This helps to mitigate the risk of misunderstandings and facilitates the resolution of issues.
– *Communications management:* Due to a globally dispersed project structure, communication paths and contact persons for all parties involved have to be defined. In very large projects, the communication structure between onshore and offshore project managers has to be established. Well-organized communication is very important. Otherwise the remote teams or team members tend to make their own assumptions and decisions which may later prove to be wrong. Extensive reworking might be needed as a consequence of this. Ongoing information and feedback on relevant issues can alleviate these communication problems. Day-to-day communication between the team members can largely be based on tools nowadays. Some of these tools will be discussed in chapter 9.
– *Risk management:* Cultural differences between onsite and offshore teams or project members deserve special attention. They are a frequent source of misunderstandings and miscommunication, leading to unsatisfactory intermediate or final project results. Risks may also be caused by different quality perceptions onsite and offshore, and by the formation of subgroups onsite that work against the offshore group for social or political reasons. The risks associated with cultural differences, different languages and subgroups are often underestimated.

8.5.2 Organization of Globally Distributed Projects

Global distribution of project work comes in many variants. There is a spectrum of options regarding how distributed work can be organized.

For simplification, we will discuss only the two endpoints of the spectrum: 1) a virtual team and 2) collaborating teams.

Virtual team

A virtual team is one team

A virtual team is a project team composed of team members from different geographical regions and/or different organizations. It has the same goals and objectives as a traditional team. However, it operates across time, geographical locations and organizational boundaries. A crucial enabler for virtual teams is information and communication technology. With the help of technology, it is possible that dispersed members can operate as a single team.

Virtual teams have become a common phenomenon in software development. For example, a medium-size team of 14 persons could be operating with a project manager in Frankfurt (Germany), three senior software developers and two quality experts in Berlin, six programmers in Hyderabad (India) and two designers in St. Pertersburg (Russia).

A virtual team will basically be organized in the same way as a non-virtual team such as the one shown in figure 8-13 above. It has one project head (or project manager) who is responsible for the project and who communicates with all team members. Virtual teams like this are better suited for small projects (or sub-projects) than for large projects.

Collaborating teams

Several teams with separate responsibilities

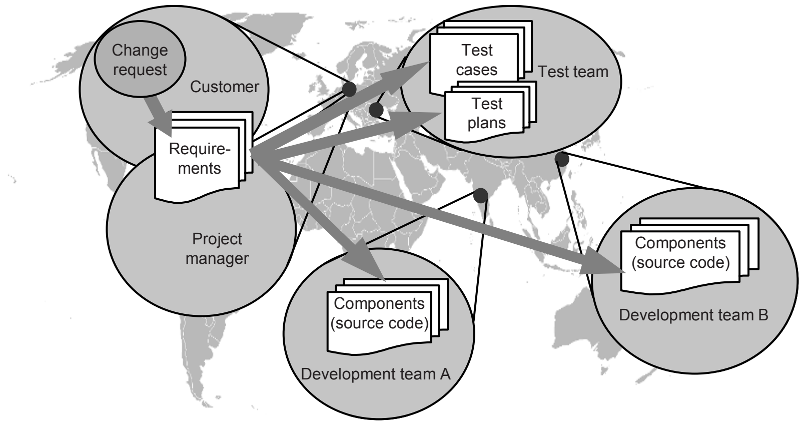
In most large projects, work is divided up among several teams with separate responsibilities. Each team is responsible for a sub-project or for a set of activities within the overall project. This means that the teams have their own project managers, contact persons and organizational structures.

Figure 8-26 illustrates this approach with the help of an example. Five dispersed teams are collaborating. The overall project management and the requirements engineers who are interfacing with the Hamburg based customer work in Berlin (Germany). The overall system has been decomposed into two subsystems A and B. These subsystems have been assigned to different development teams, one in Bangalore (India) and one in Shanghai (China). Integration and testing is done by a test team in Bucharest (Romania).

Onsite-offshore coordination

In a distributed project organization like the above, it is important to have well-defined interfaces between the central project management and the decentralized teams. Many companies practicing offshoring set up one or more roles for onsite-offshore coordinators.

Figure 8-26 **Global software development**



A typical example for a medium-size project is shown in figure 8-27. On the left-hand side, the onsite project team is shown. The central project management is reinforced by an *offshoring manager* and an *offshoring software engineer*. The offshoring manager is responsible for planning, controlling and coordinating cross-location work activities, including the resolution of conflicts and intercultural issues. The offshoring software engineer is in charge of technical coordination, in particular answering technical questions regarding specifications, documents and deliverables and clearing up technical misunderstandings.

Offshoring manager, offshoring software engineer

The onshore company also has an *offshore coordinator* at the site of the offshore company (or captive center). This person serves as a contact to the local team for technical and administrative questions. He or she helps to avoid misinterpretations and the rise of issues that would otherwise escalate.

Offshore coordinator

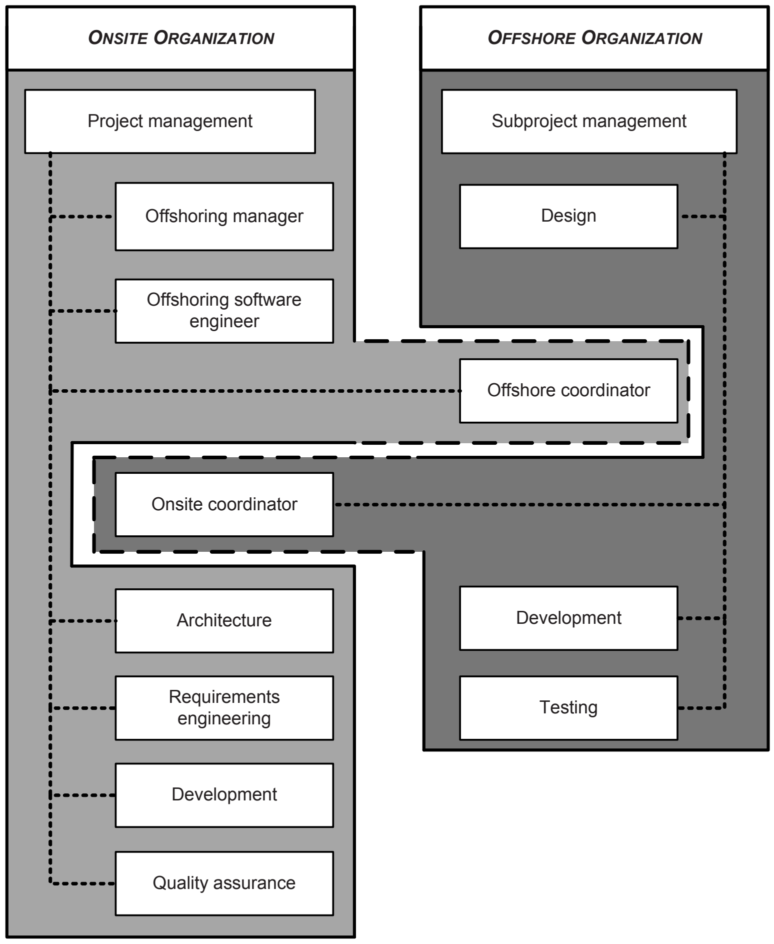
Likewise, the offshoring provider has a coordinator working onsite *(onsite coordinator)*. This person's task is similar to the offshore coordinator's task – clearing up open questions and resolving technical issues with the onsite personnel.

Onsite coordinator

A process-oriented view of the roles and interactions between the two organizations involved is shown in figure 8-28. In this example, the central team (on the left) provides the system requirements, architecture, design, development plan and acceptance tests for components to be developed offshore.

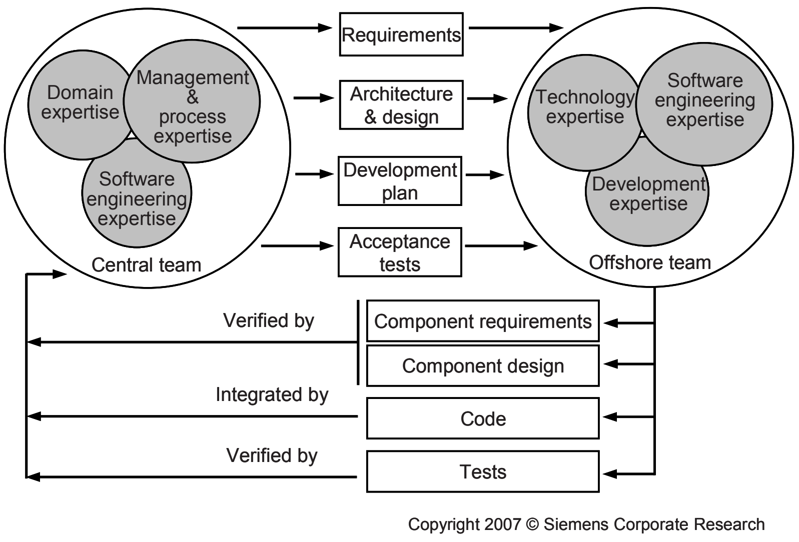
Interaction between onsite and offshore organizations

Figure 8-27 **Interfacing onsite and offshore organizations in ISD**



The offshore partner specifies the component requirements and designs which are verified by the central team. Module integration is also done by the central team. Testing is a task of the offshore team, prepared by the central team that prepares acceptance criteria and verifies the tests performed offshore.

Figure 8-28 **ISD processes involving remote teams [Paulish 2007]**



An earlier example of onsite and offshore tasks was given in the discussion of offshoring process models (cf. section 4.5.1). In figure 4-23 of that section, we illustrated the responsibilities of onsite and offshore partners along a sequential process model.

9 Tooling Support for ISD

by Brandon Ulrich

A typical information-system development project today is characterized by team work where team members are not necessarily collocated, but possibly distributed around the globe. This has led to increased demands regarding collaboration. To enable this collaboration, effective tooling needs to be in place to support the development processes.

Fundamental, combined and project-wide tools

As fundamental tools have stabilized and become commodities, the focus on tooling improvement for individual developers has shifted to teams of developers and now to geographically distributed development teams. Tools make it possible to address the core requirements of a distributed development environment, make the status of a project transparent to all participants, and help to disperse knowledge from individual team members.

In this chapter, we will discuss three primary areas of tools that assist with global software development projects. The first are fundamental tools, which provide the necessary foundation to build a development process. Based on these fundamental tools are additional tools that support the automation of projects. Finally, once the projects have been automated, tools for task-focused, context-based and process-focused development try to increase the productivity of a distributed team.

9.1 Fundamental Tools

Three fundamental types of tools are critical to the success of any software project: source control systems, defect tracking systems and testing tools. Source control systems maintain the history of code used in a software project. Defect and issue tracking systems record problems and enhancements for a system in a structured manner. Test tools help ensure that code is correctly functioning. Together, these sets of tools provide a foundation onto which higher value added project services can be built.

In this section, we will present source-control, defect-tracking and test tools (load-testing tools). Since the other type of test tools (unit-testing tools) were discussed before, in the context of testing issues (cf. section 6.3), only load-testing tools are included below.

9.1.1 Software Version Control Systems

Software version control systems help a team manage the source code used throughout the development process. The version control systems allow distributed teams of developers to collaborate on the same projects.

Maintaining a history of code

Many terms exist for the concept of maintaining a history of code within a software development project, including: revision control, version control, source (code) control, source code management, as well as configuration management, which covers a wider area. However these ideas have the same underlying concept – that each version of a software development artifact has a history that can be returned to at any time. For example, it is possible to recreate the base of source code as it appeared at any particular date and time within the project's life.

Maintaining this software history is important for many reasons. It makes it easy to undo mistakes. It allows multiple developers to work on a body of source code in a controlled way. Version control also allows auditing and metrics of the development process and can be "invaluable for bug-tracking, audit, performance, and quality purposes." [Hunt 1999, p. 83] By allowing the creation of any prior release, it allows multiple versions of a software product to be supported.

Project repository

A version control system uses a *repository* as the central storage area for all versions of the project's files. A *version* is a certain set of files at a particular point in time. Before developers work with a repository, they must *check out* the source code. This collection of source code on the developer's machine is called the *working copy* [Mason 2006, p. 12]. After editing it, they update, or synchronize their code with the repository to identify any changes that have occurred while they were editing the source code on their local machine.

Committing changes to the repository

After updating, developers *commit* their changes to the repository so that they are available for other developers. When using systems that restrict editing to one developer at a time (discussed below), a developer will *check in* their code after they have finished working on it. It is considered good practice for a developer to commit their code as soon as they have completed the task that they are working on [Subramaniam 2006, p. 162].

Version control systems allow the undoing of mistakes since it is always possible to return to a previously working state at any time. If a series of code changes caused a problem, these can be easily reverted so that the code is working as it was before.

Concurrent editing support

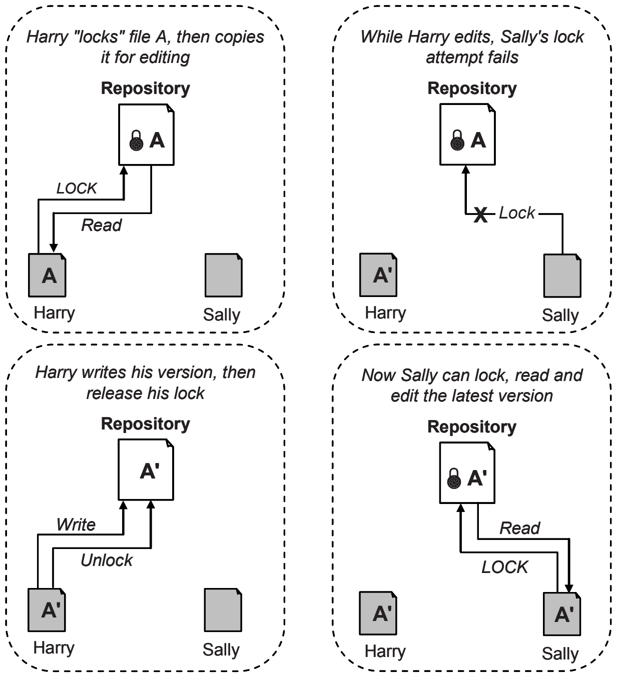
Multiple developers work on the source code

One of the biggest advantages of version control systems is that they allow multiple developers to work on the same collection of source code simultaneously. This means that the development team can continue to work with and make changes to the code while their teammates are working on different areas. To provide for concurrent access, version control systems support two primary types of versioning models, either locking or version merging.

File locking

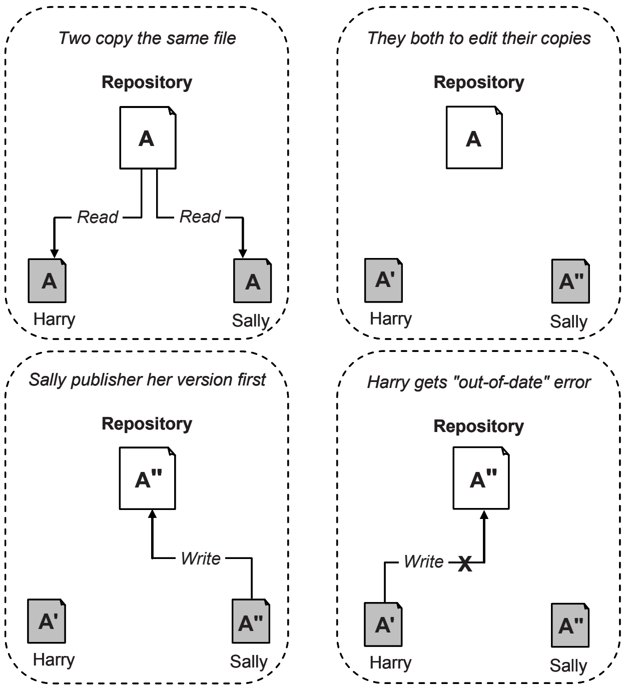
File locking is relatively straightforward: The requested file is simply locked until the developer has finished work on it. During that time, no other developer can modify the file. After the developer has finished updating the file, the new version is added to the source code repository and the file lock is removed. Although simple, a locking system is more useful to smaller development teams working in close geographical proximity to facilitate collaboratively editing files. With more distributed teams, a frequent challenge is deciding whether or not to override a colleague's lock – especially considering that they might be located in a time zone 8 hours ahead.

Figure 9-1 File locking [Collins 2004, p. 4]


Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A'
Harry
Sally
Harry writes his version, then release his lock
Repository
A'
Write
Unlock
A'
Harry
Sally
Now Sally can lock, read and edit the latest version
Repository
A'
Read
LOCK
A'
Harry
A'
Sally

The second model for version management is *version merging*. When using version merging, there are no file locks. Instead, multiple developers may modify the file at the same time. Although this may sound like a recipe for disaster, in practice, most modifications to a file are localized and can be automatically merged. For example, if developer A modifies a method called "equals" at the same time that developer B modifies a method called "toString", these changes can be automatically merged. The automatic merge works by requiring that all developers update (i.e. synchronize) their code with the repository prior to committing their changes. In the above example, when developer B updates her code, she will receive developer A's changes to the "equals" method. In the case that multiple developers are working on the same area of code at the same time, a conflict occurs that must be manually merged by the developer.

Figure 9-2 Version merging [Collins 2004, p. 5]



Multiple software version support

Generally there are multiple versions of software in use at any one given time. A company may release a 1.0 version of its software, which is adopted by several customers, followed later by several smaller "point" releases such as 1.1, 1.2, and 1.3. At any time, different customers may be using different versions of the software. If an important customer identifies a problem in the 1.1 release, it will become necessary to fix the problem in the code for that specific release.

Multiple versions in use at the same time

Because development of a product does not generally stop after a release, it is quite common for the development team to be working with a completely different set of files than those currently in use by the customer. A version control system lets the team return to the software source files as they appeared to the customer in a previous release. They can then fix the problem in the version that it occurred. The fix may need to be moved forward to the main line of development or other supported branches.

Common version control systems

There are a large number of commercial and open-source version control systems. The two most popular ones are Concurrent Versions System and Subversion.

Concurrent Version System (CVS)

The *Concurrent Version System* (or Concurrent Versioning System) is most commonly known by its abbreviation, *CVS*. It was created in 1985, is freely available as open-source, and has a large installed base of users. CVS is a client-server system, with a CVS server that contains the source code and the change history and a client that connects to the server.

As the "concurrent" in its name implies, CVS allows multiple developers to work on the same file at the same time via their CVS client, instead of requiring an explicit file lock. Developers update their code to synchronize their changes with the server, and then commit their changes to the server after testing.

Subversion (SVN)

Subversion is a more recent version control system, also commonly known by the initials of its client program, *SVN*. It was first released in 2004 and is an open-source tool that shares many features with CVS. The number of SVN users has been steadily increasing. Although reliable usage statistics are difficult to obtain, since companies frequently use SVN and CVS for internal development projects, SVN usage now appears to be more popular than CVS for open-source development projects (http://cia.vc/stats/vcs).

9.1.2 Issue and Defect Tracking Systems

Issue and defect tracking systems are used to maintain the list of problems found within a product. They help organize development projects by maintaining a database of issues with the software. They also provide a means of scheduling work across a development team, since the issues and defects can be assigned to specific developers or distributed teams.

An issue is a defect or bug

An *issue* is a defect or bug found in the software – issue tracking systems are also called defect tracking systems or bug tracking systems. By maintaining a list of the defects in one place, management of the corrections becomes simpler, workarounds to known problems can be documented, and prioritization of the issues is transparent.

The information recorded by a defect tracking system depends on the system, its configuration and the organizational requirements. However, there are common sets of information necessary to track regarding all issues. First, which product is affected by the problem? What component of this product is involved? What version of the product is affected? Are other versions also affected? Who discovered the issue? What are the required steps to reproduce the problem?

Defects can be prioritized

Issue tracking systems also allow a defect to be prioritized. Some systems support both *severity* and *priority* concepts, since the priority of a defect is often based on business reasons rather than technical reasons. As an example, a high severity item that causes a major system malfunction but is only rarely encountered may have a lower priority than a less severe bug that is encountered by most customers [Rothmann 2005, p. 80].

Once this basic information is recorded, more detailed information may be added as further research is done on the issue. In the end, the issue will be marked as fixed, left open or recorded as unable to reproduce. When the issue is fixed, it is important to record the developer that completed the fix and the version of the software that includes the fix.

Having all of this information in a defect tracking system allows support personnel and users to query the database of problems to see if the problem that they have encountered has already been identified. If so, they can find a workaround, if it exists. They can also add more detailed information about the problem that they identified.

Reporting on issues

Issue tracking systems also include reporting systems generating reports of the issues. This can give a good picture of the overall quality of a product version. Reports are also useful to identify trends in the quality of software and to take steps to improve the quality if necessary. Examining these reports to identify the number of issues entered, the number of issues fixed, and the amount of time necessary to fix them can help identify trouble spots in a project [Richardson 2005, p. 36].

Features and enhancements can also be included in issue tracking systems. This allows the prioritization of new work that will be performed. By including features and enhancements, issue tracking systems adopt characteristics of task-management systems.

Example defect life cycle

The process of entering and resolving a defect has a life cycle. This life cycle is similar between projects and tools. An example of a defect life cycle for the Bugzilla issue tracking system is shown in figure 9-3.

A typical path for a defect through this life cycle is as follows. When a new defect is entered into the system, it is in the UNCONFIRMED state until the presence of the bug is confirmed. The bug remains in the NEW state until it is assigned to a developer.

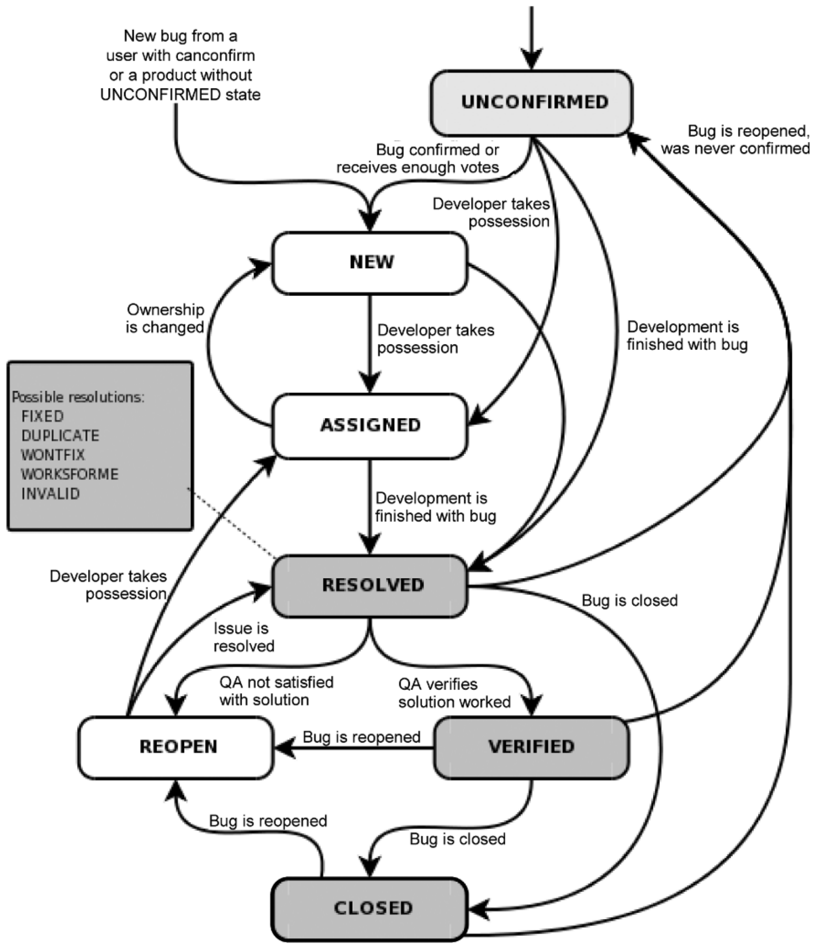
Once the issue is assigned to the developer, the developer has several ways of resolving the defect. The developer may fix the code responsible for introducing the defect. A developer might also mark the bug report as a duplicate of an existing report. If the developer cannot reproduce the problem, they mark it as WORKSFORME. Otherwise, the developer may mark the bug as INVALID if something is wrong with the report or as WONTFIX if the defect will not be corrected.

After the bug has been resolved, it may be verified by either the reporter or by a quality assurance department. If the bug has not been fixed, it will be reopened, otherwise it is closed.

Common issue tracking systems

There are many issue tracking systems, both commercial and open-source. Of the freely available open-source systems, *Bugzilla* is the most popular. It was developed in 1998. Bugzilla is written in Perl and runs on a web server such as Apache. It requires a supported database, such as MySQL or PostgreSQL. An example of Bugzilla use in a project can be seen at the Eclipse website (https://bugs.eclipse.org/bugs/).

Figure 9-3 Bugzilla life cycle [Mozilla 2007]


New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
Possible resolutions: FIXED DUPLICATE WONTFIX WORKSFORME INVALID
ASSIGNED
Development is finished with bug
Developer takes possession
RESOLVED
Bug is closed
Issue is resolved
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

JIRA is a popular commercial product that provides open-source projects a free license to use it. It has more features than Bugzilla, and supports tight integration between source code control systems and the issue tracking system. Reporting is also well supported. An example of a JIRA system can be seen at the JBoss Seam website (http://jira.jboss.org/jira/browse/JBSEAM).

9.1.3 Load-testing Tools

A load test is a stress test

Load-testing tools are designed to ensure that an application, usually a server-based application, can reliably support a specified number of users. The number of users is referred to as the *load*. A load test is a stress test on the server, used to determine if the server will respond within a certain response time, or fail.

How many users can be supported?

A common use of a load test is to determine the maximum number of concurrent users that are supported by a certain hardware configuration. Once the server application has been installed, the load test tools simulate an increasing number of users until either the system breaks or the response time becomes unacceptably high.

For example, consider load testing an auction site. If the developer wanted to determine the maximum number of concurrent users that can perform a search, they could set this up as a single test. Then the load testing tools will run an increasing number of these users together, recording the response, the number of concurrent users and the response time. At some point in the stress test, either the response will be erroneous due to a software failure or the response time will increase to a point considered to be a failure (e.g. 5 seconds). The number of concurrent users then registered is regarded as the number of concurrent users supported by this particular test.

However, users exhibit different patterns of activity, which would not be correctly indicated in the above test. In the auction site example, some users would be searching for items, some would be bidding on items, some would be entering new auctions, and others would be leaving feedback or arranging shipping after the conclusion of an auction. To get a realistic real-world picture of the number of users supported, it is necessary to set up scenarios that represent real-world use.

Individual usage scenarios

These individual usage scenarios are then executed in parallel with the help of a load-testing tool. The proportion of scenarios being executed can be set at specific levels, varied randomly, or usually fluctuating between real-world values. In the auction site example, a user might execute a search, look at the results of that search for 3 seconds, click on a search result to see the item, read about the item for 12 seconds, and then place a bid on the item. A more typical case might omit the end-of-cycle purchases, just repeating user searches and item browsing.

A load-testing tool would allow parallel execution of these scenarios to give a typical result of the number of actual users that can be supported by the system. Real-world use is much less demanding than concurrent uses, because pauses in the users' behavior are reflected in the scenarios. This translates to a more realistic picture of the number of actual users that would be supported on a particular hardware platform.

In practice, both types of testing are valuable. Although real-world usage tends to be of the most interest for scaling a system, it is also nice to know the point where the system will unequivocally fail. Returning to the auction-site example, it would be useful to know the maximum number of users that can submit a bid on an item at the same time, since many users wait until as near as possible to the end of an auction to bid.

Common load testing tools

There are a variety of commercial and open-source load testing tools. Many small projects simply write unit test code to measure the results themselves. Larger projects use commercial tools such as Mercury Load Runner[§] or open-source tools such as Apache JMeter [Apache 2008].

9.2 Combining Fundamental Tools

This section discusses tools that build upon the fundamental tools to provide additional value to an ISD project. These tools are release engineering tools, quality control tools and continuous integration tools.

9.2.1 Release Engineering Tools

Based on software version control systems

Release engineering tools use the software version control systems discussed in section 9.1.1 in combination with testing tools to automate the process of constructing a version of software. Automating this pro-

§ http://www.mercury.com/us/products/performance-center/loadrunner/.

cess allows developers working in distributed teams to easily reproduce the software assembly process carried out by their distributed peers in a controlled and repeatable manner.

"Build"

In software development organizations, a "build" is a term for constructing a version of software. Release engineering tools are more frequently called *build tools* and are designed to automate the process of creating software, mostly by automating the series of tasks necessary to compile, test and deploy software systems.

Automated software builds

Automated software builds have removed the huge integration builds at the end of a development cycle. Instead of creating the final build at the end of the cycle and "throwing it over the wall" to the quality-assurance teams, complete builds and tests are run more frequently. Over time, the frequency has moved from weekly builds to daily ("nightly") builds. The recent introduction of continuous integration tools (see below) now allow builds and automated tests to be run every time a line of code is changed.

The build runs in a series of dependent stages. A failure at any of the stages means that the build will terminate abnormally. Before the build can run, the appropriate source code must be retrieved from the source code repository (see section 2.2). This may be the most recent version (the "mainline"), or it might be a historical version that is maintained within the source repository.

Compiling and testing

The first step of the build tool is to compile the necessary code to make the system. Simple systems may be composed of modules written in a single language; in this case, the same compiler is used for all modules. More complex systems may use a variety of languages within and between modules and run on diverse platforms (e.g. Windows, Mac and Linux platforms). Once the project has been compiled, automated test tools are generally run by the build script. Again, multiple platform support can add complexity to the build process.

Packaging

After the tests have been run, the software product may be packaged. For example, Java projects which will be deployed on the server need to be assembled in a certain way. Support and configuration files must be copied to the correct locations. Documentation and on-line support specific to the local language must be built and packaged.

Deploying

After packaging is complete, the package may be deployed. Depending on the type of application, this might mean running the installation routines created in the packaging stage. Server applications must be deployed on one or more target servers. This step often includes database deployment and set up.

Advantages of an automated build system

A manual build system requires a developer to go through a series of steps. These steps increase dramatically as different languages and platforms are supported. Complex systems that require testing on several application servers or databases involve so many steps that the ability to reproduce a build by following manual processes becomes suspect.

By automating the build, it becomes possible to construct a version of the system as it was at any time in the past. Retrieving the relevant source code from the repository before the build runs means that the build can be reproduced as necessary for any version of software. Since the build configuration files themselves are usually included within the source code repository, changes to the build over time are maintained.

Additionally, individual developers can also construct the software on their own system to ensure that changes they introduced did not adversely affect the overall system stability.

Common release engineering tools

There are a number of release engineering tools available. They depend upon the system being built. *Make* is popular for C/C++. *Apache Ant* is a very common and freely available open-source tool for Java. Ant runs on any platform that supports a Java virtual machine. Ant uses a build file to control the process of the build. The build is divided into separate tasks, which do things such as compile code, create a zip, copy files etc. Ant is also designed to be extensible, so it is possible to create customized Ant tasks to meet requirements that were not foreseen by the original authors.

9.2.2 Quality Control Tools

Quality control tools ensure that standards in coding, documentation and quality control are handled consistently across development teams. They can also detect potential programming problems, design issues and legal IP (intellectual property) problems. These tools may run at different stages of the build process. For example, some tools focus only on source code, whereas others inspect the object or machine code to identify potential issues.

Enforcing coding standards

There is a great deal of flexibility in the way developers write source code. Different standards for naming files, modules, packages, methods, variables etc. can exist. The organization of a file is also up to the developer writing the code. How code is indented, how lines are wrapped, which line length to use are all potential sources of inconsistency between developers. Other formatting issues of interest are commenting standards, programming statement organization and declarations.

Naming conventions

Consider a simple question of naming conventions. Within the same set of coding conventions, different names may imply different things. For example, in the code conventions for the Java programming language [Sun 1999], the usage of case implies meaning. FooBar, fooBar and FOO_BAR, for example, each represent different things. Beginning each word with a capital, such as "FooBar" meets the standard for a Java class name. A name that is completely capitalized with an underscore separating words in the name, such as FOO_BAR, indicates that FOO_BAR is a constant. Names that begin with a lowercase character, such as fooBar, are used to indicate variable names.

Compilers do not enforce conventions

However, these are merely conventions, and are not enforced by the compiler in any way. This can allow developers to use different standards, or simply to make mistakes in their naming that go against the standard. When a different developer expecting to be reading code that abides by the standard reads the first developer's code, they will be confused or make incorrect assumptions about the code. In either case, precious time is lost in understanding and reading the code.

To support developers who wish to abide by these standards, development tools usually include the ability to format code according to a specific standard. For example, the Eclipse IDE supports a large number of coding standards and can automatically reformat code to comply with these standards. An example of supported conventions related to organizing control statements in the Eclipse IDE can be seen in figure 9-4.

Tools that verify the conventions

However, these tools are only successful if used by each of the developers. Since developers use a variety of program editors, there may be differences in the support for maintaining the conventions. Therefore external tools that can be integrated into a build process are frequently used to verify and maintain the conventions across an entire project.

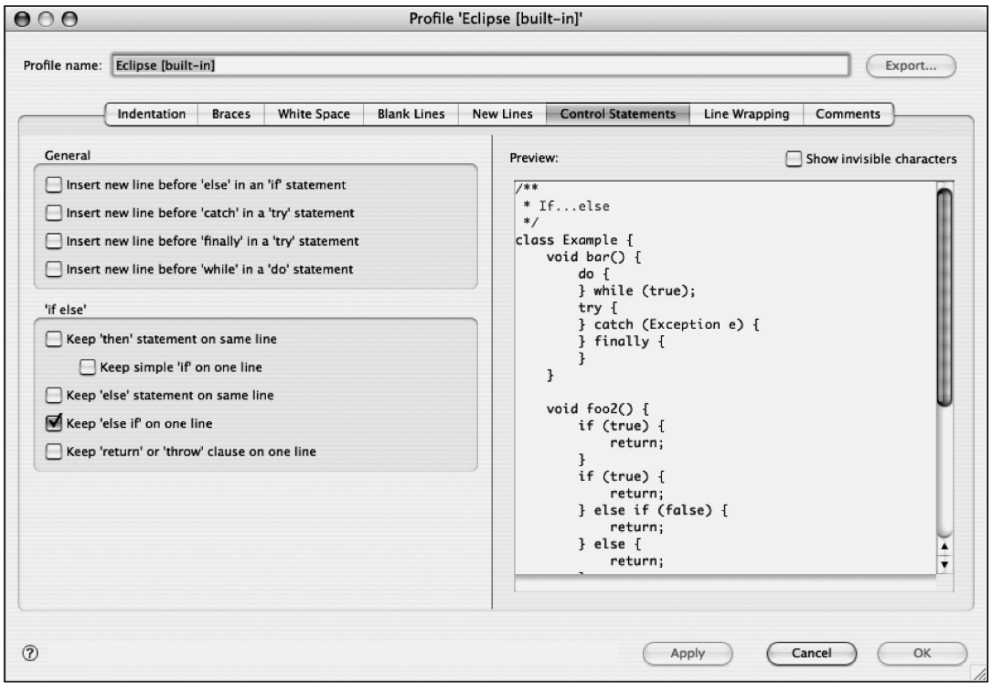
An example of such a tool is *CheckStyle* [CheckStyle 2007]. It is freely available as open-source and designed to verify that Java programs adhere to a particular coding standard. CheckStyle can also be integrated into an automated build process.

Checking documentation against code

An advantage of modern programming languages such as C# and Java is that they allow a developer to include documentation directly within the source code. C# allows developers to embed XML comments that can be exported and formatted. Java provides *Javadoc*, which supports adding comments to code. A processor can then go through the code and build XHTML documentation for the code.

This documentation is important, not only as an external reference to understand the options for calling routines and their meanings.

Figure 9-4 Code convention support in the Eclipse IDE



Profile name: Eclipse [built-in]

General

- ☐ Insert new line before 'else' in an 'if' statement
- ☐ Insert new line before 'catch' in a 'try' statement
- ☐ Insert new line before 'finally' in a 'try' statement
- ☐ Insert new line before 'while' in a 'do' statement

'if else'

- ☐ Keep 'then' statement on same line
  - ☐ Keep simple 'if' on one line
- ☐ Keep 'else' statement on same line
- ☑ Keep 'else if' on one line
- ☐ Keep 'return' or 'throw' clause on one line

Preview: ☐ Show invisible characters

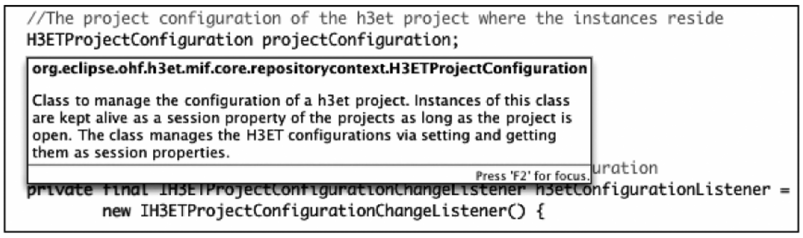
```
/**
 * If...else
 */
class Example {
    void bar() {
        do {
        } while (true);
        try {
        } catch (Exception e) {
        } finally {
        }
    }

    void foo2() {
        if (true) {
            return;
        }
        if (true) {
            return;
        } else if (false) {
            return;
        } else {
            return;
```

Modern IDEs also include the ability to automatically prompt the programmer with this documentation. This support is often displayed after a certain time; for example, after the cursor hovers over a method call for a particular period of time or when a developer pauses before entering parameters in a Java class constructor. An example of this support in the Eclipse IDE can be seen in figure 9-5. For distributed team members that often write code based solely on the published API of modules developed by other teams, documentation can become a primary means of communication.

Documentation – a means of communication

Figure 9-5 Javadoc tooltip within Eclipse IDE



```
//The project configuration of the h3et project where the instances reside
H3ETProjectConfiguration projectConfiguration;
```

org.eclipse.ohf.h3et.mif.core.repositorycontext.H3ETProjectConfiguration

Class to manage the configuration of a h3et project. Instances of this class are kept alive as a session property of the projects as long as the project is open. The class manages the H3ET configurations via setting and getting them as session properties.

Press 'F2' for focus.

```
                                                        uration
private final IH3ETProjectConfigurationChangeListener h3etConfigurationListener =
        new IH3ETProjectConfigurationChangeListener() {
```

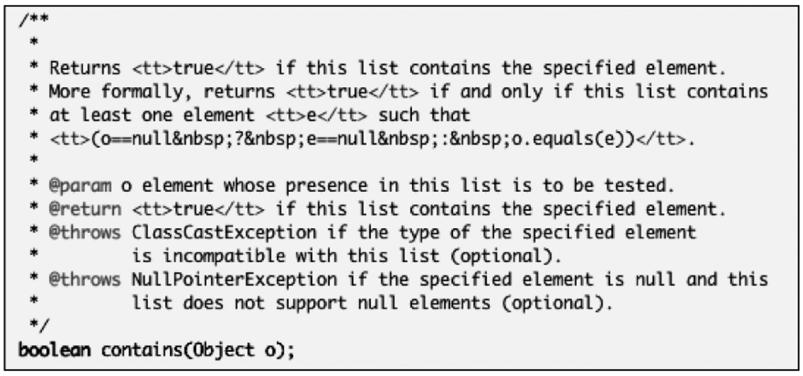
Documentation should match the source code

Although there are clearly many advantages to documenting code directly in the source code files, it can be challenging to maintain the documentation so that it matches the source code. To understand this, it helps to look at an example of a Javadoc comment for a method. Figure 9-6 shows an example Javadoc for a method named "contains".

Javadoc

For comparison, figure 9-7 shows how the documentation code from figure 9-6 would be rendered in display to a developer. In the documentation, it is clear that some elements are rendered differently. For example, <tt>true</tt> renders the text "true" using fixed-width text ("typewriter text"). There are several other formatting commands similar to HTML.

Figure 9-6 Javadoc for "contains" method



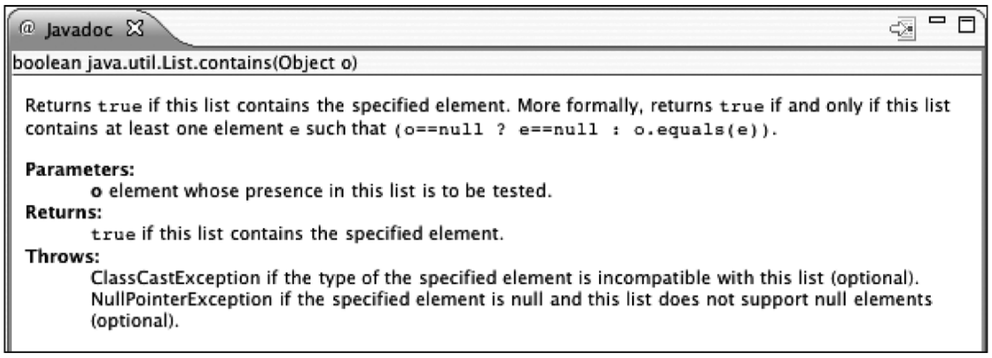
```
/**
 *
 * Returns <tt>true</tt> if this list contains the specified element.
 * More formally, returns <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null&nbsp;?&nbsp;e==null&nbsp;:&nbsp;o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is to be tested.
 * @return <tt>true</tt> if this list contains the specified element.
 * @throws ClassCastException if the type of the specified element
 *         is incompatible with this list (optional).
 * @throws NullPointerException if the specified element is null and this
 *         list does not support null elements (optional).
 */
boolean contains(Object o);
```

There are also some tags referred to as *annotations*. Here @param, @returns and @throws are used to respectively indicate the method parameters, the return class and any exceptions that the method call may throw. Note that although the parameter name "o" for an object ("@param o element…") is intended to match the "o" found in the method call ("contains(Object o)"). However, there is no compile time checking that this is the case, so there is the possibility that errors may be introduced by a careless developer making changes to the parameters of a method.

Documentation errors

Since these documentation errors will show up throughout the development environment and in the published documentation, these inconsistencies can lead to wasted time as developers try to understand why a routine works differently than documented. The integrated IDE support for the documentation also makes the documentation appear as more than simple text entered haphazardly by a distracted member of the development team.

Figure 9-7 Formatted Javadoc



boolean java.util.List.contains(Object o)

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Parameters:
 o element whose presence in this list is to be tested.
Returns:
 `true` if this list contains the specified element.
Throws:
 ClassCastException if the type of the specified element is incompatible with this list (optional).
 NullPointerException if the specified element is null and this list does not support null elements (optional).

DoctorJ

There are various tools available to help detect consistency problems between the documentation and the code. *DoctorJ* [Incava 2008] is a freely available open-source example of such a tool. It can detect problems between missing, misspelled or incorrectly ordered parameters and exception names, and help to identify a developer modifying the code without updating the documentation. It can also identify problems with incorrect Javadoc tags and indicate if documentation is missing for a class.

By incorporating one of these tools within the build process (see section 9.2.1), documentation errors can be identified before they influence productivity.

Cut-and-paste code detection

Redundancy, legal and IP problems

Problems can occur when code is copied and pasted between two source files. This can indicate poor planning or team coordination [Gurses 2005]. Code should be refactored to avoid the problems of maintaining identical code in multiple locations. Furthermore, duplicated code may imply legal and IP (intellectual property) problems – even in open-source projects. This is because each source artifact has an associated copyright, even if the license is open.

In large projects with work broken between many different teams, it is often simpler for a hurried developer to copy code from a different team's project. This may be because of bureaucratic or legal obstacles between team collaboration, or simply due to time constraints. However, reusing the borrowed code may have viral IP effects on the product the developer is creating.

Cut-and-paste code detectors can help identify these problems before they become an issue. These tools use a variety of algorithms to identify code that has been "borrowed" from external sources.

PMD (there is no meaning associated with the letters; cf. http://pmd.sourceforge.net/) is a freely available open-source tool. It has a cut-and-paste detector to identify suspect code, that can be easily integrated into a build process.

9.2.3 Continuous Integration Tools

Combining several fundamental tools

Continuous integration tools combine several fundamental tools and techniques described in the previous sections. Continuous integration tools are used to increase the number of automated builds to a practically "continuous" basis, ensuring that everyone on a team, regardless of their location, has access to the most recent build. While automated builds are frequently scheduled to run on a weekly or even a nightly basis, continuous integration tools trigger an automated build more frequently – ideally whenever code changes. "Many continuous integration tools use a "polling" strategy to check whether anything has changed within a repository, and if it has, to automatically do something useful like building the latest code and running tests." [Mason 2006, p. 160]

Continuous integration works with version control systems, release engineering tools, bug-tracking systems and software quality tools. Of these, the idea of automatically testing the build is paramount: "Continuous integration assumes a high degree of tests which are automated into the software …" [Fowler 2006]

Triggering builds automatically

Once an automated build process has been established and a build can be created at any time with a simple command-line call, it becomes easier to add more advanced functionality. This is where continuous integration comes into play. The theory is that a build should be *automatically* triggered whenever there is a change to a project's source code.

Need for frequent releases

The driver for increased use of continuous integration tools is the need for more frequent releases. As development cycles have decreased in length, the amount of time that a team has to focus solely on system integration and integration testing has decreased. This has led to more frequent internal releases which are intended for the development team itself, as opposed to external customers.

Benefits of continuous integration

Risk reduction

Of the many advantages of continuous integration, Fowler believes that risk reduction is the "greatest and most wide ranging benefit." [Fowler 2006] The risk is reduced because integration problems are addressed immediately, instead of at the end of a release cycle.

The immediate feedback provided by identifying bugs also helps to decrease the amount of time necessary to fix the bug. This is because when a new build has introduced a bug, a developer can compare the current build to the previous build. By comparing the source used in both of these builds, isolating the breaking change becomes much simpler.

More confidence

Another advantage of continuous integration is the confidence with which developers can make larger changes to code. In many development projects, there is a particularly complex area of logic that the team is hesitant to change because of numerous dependencies. Instead of refactoring this area of code to improve the quality and reduce the complexity, the fear of breaking dependent systems means that the code looks more and more complex. Applying a continuous integration build can provide the confidence that dependent problems will be automatically and immediately identified – giving the developer the confidence to improve the code.

Reporting and monitoring

Information reporting and monitoring is another advantage of continuous integration systems. The most obvious way to report a build failure to the team is by the use of an e-mail, monitor, light or other device.

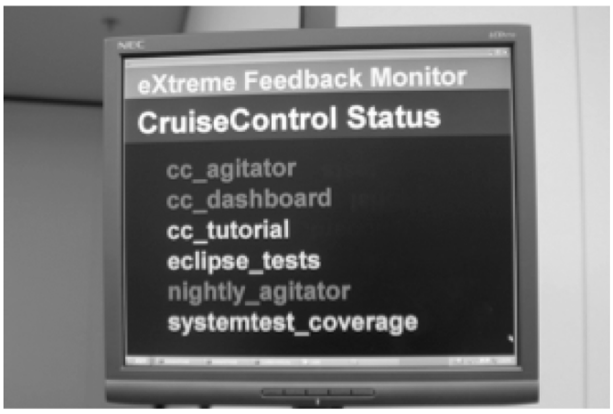
E-mail notification of build problems means that some or all of a team will be notified as soon as an event occurs. A typical configuration is that an e-mail is sent to the person that made the breaking change in the source code, identified by their user identification in the source code control repository. E-mails may be additionally sent to the entire team, or simply the project manager.

Instant messaging notification is also popular. Build failures can also be sent via test messaging to the mobile phones of those affected [Clark 2004, p. 126].

Notification mechanisms

However, clear visual indicators of the current build status are very important. Often a computer monitor is dedicated to the task of displaying the status of the build for the entire team (cf. figure 9-8). This allows everyone to see at a glance that everything is running smoothly. When a build fails, this machine may be configured to sound a tone, play a song, or provide similar audible feedback.

Figure 9-8 Continuous integration feedback [Clark 2004, p. 130]



Other notification mechanisms use an X10 module (a device that reacts to a radio signal to turn on or off any electrical device). For this to work, the continuous integration machine is equipped with an X10 transmitter, which switches on or off electrical devices to indicate the build status, e.g. a green or a red light to indicate that the build is successful or unsuccessful [Duvall 2007, p. 216].

Ambient Orb

While popular and inexpensive, X10 notification methods have the disadvantage that they are binary information; they are either on or off. An increasingly popular tool for visual notification is a product called the *Ambient Orb* (http://www.ambientdevices.com/cat/orb/). This is a frosted-glass ball that glows different colors to display real-time information. This device receives radio signals that allows it to vary its color. This ability can be used to indicate ranges, such as the level of code quality during a particular continuous integration build. When the code base is at a high quality the light is green, as quality deteriorates, the color of the ball changes to lighter shades of green, then yellows, and eventually to red.

Common tools

Continuous integration tools are available for a wide variety of platforms and languages. *CruiseControl* is the most popular choice for Java projects. It was developed by Martin Fowler's company for an internal project and is now freely available as open-source (http://cruisecontrol.sourceforge.net). *AntHill* is another popular continuous integration tool on the Java platform available in both commercial and open-source versions (http://www.anthillpro.com).

Example of continuous integration tool use

This section will discuss the use of a continuous integration tool using *CruiseControl* as an example. We will discuss the workflow from the perspective of a developer working on a project that is using a continuous integration system.

Checking out the project

Developers working on a project using source code control start by checking out the project from the source code repository. These steps and examples were discussed in section 9.1.1. After the developer has a local copy of the source code, they make changes to the source code to fix problems or add additional functionality.

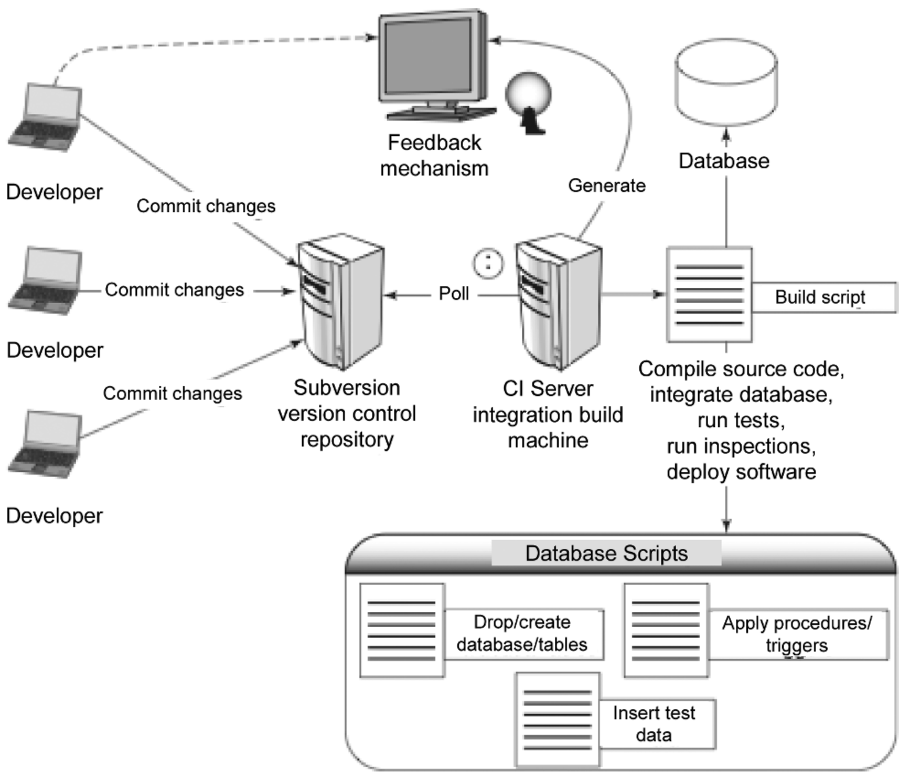
Once the code is complete, the developer will run their suite of automated tests on the code that contains their changes. This ensures that they have not introduced new problems into the code while making their changes.

Of course, the developer is not working in isolation; other developers may have made changes while the first developer was completing their work. Therefore, after the local tests are successful, the developer updates their source with any changes from the repository. If there were new changes, the developer runs the tests again to verify that his or her code works correctly with the latest source. Once the tests pass, the developer commits the changes to the repository (cf. figure 9-9).

Continuous integration server

This is the point where the continuous integration server comes into play. A continuous integration system is usually configured to poll the source code repository for changes. The length may vary depending on how long the actual build takes to complete on the server. If the continuous integration build takes a few minutes, as may be the case with a large set of integration tests, then the continuous integration build might be configured to run every 5 minutes or so.

Figure 9-9 Continuous integration process [Duvall 2007, p. 15]


Feedback mechanism
Database
Generate
Developer
Commit changes
Commit changes
Poll
Build script
Developer
Commit changes
Subversion version control repository
CI Server integration build machine
Compile source code, integrate database, run tests, run inspections, deploy software
Developer
Database Scripts
Drop/create database/tables
Apply procedures/ triggers
Insert test data

"Quiet period"

It is also important not to initiate the build process the exact second that code has been committed to the repository. This is because updates to the repository are often grouped by a developer. For example, work may be committed using different comments and therefore in different commit groups. To handle this, a so-called "quiet period" is defined so that, for example, at least 30 seconds must pass without any additional source code changes.

Once the build is initiated, the continuous integration server pulls the source from the source code repository. It then runs the build process, which generally starts by compiling the source code. Once the source code has been compiled, a set of tests is run on the compiled code to verify that no new bugs have been introduced during the last change.

Notifications

A failure may occur at any point during the continuous integration process. When this occurs, CruiseControl notifies the parties that have been specified in the configuration. Configuration options are extensive, allowing only the parties that made changes included in the build to be notified. Frequently, a failed notice may go to the project manager or perhaps the entire project team. Successful build notifications are often sent only to the developers that have made the changes.

Alternative notification mechanisms are also used, including instant messaging and mobile phone messaging. To avoid being burned out by the large numbers of changes, it is important to limit these notifications.

9.3 Project-wide Tools

This chapter discusses tools that help the development process at the higher level, by offering task-focused development, context-based development and process-focused development. These tools enable distributed groups of developers to collaborate more effectively.

9.3.1 Task-focused Development Tools

Task-focused development tools, in combination with issue tracking systems discussed in section 9.1.2, allow distributed groups of developers to focus on completing their work without affecting other developers. Most developers are required to work on several, possibly quite different, tasks during the course of their workday. In addition to working on the latest feature or product release that they are developing, they also maintain responsibility for repairing previous versions of a product or address newly found and critical bugs.

Tasks come from many sources

Developers receive their tasks from a myriad of sources. Incoming e-mails from project managers or end-users, or bugs entered in issue tracking systems may each represent a task that the developer needs to schedule and prioritize. Unless these tasks are entered in a developer's personal to-do list, they can be overlooked or missed.

Prioritizing and scheduling tasks

Task-focused development tools make use of task repositories such as issue tracking systems (cf. Bugzilla or JIRA mentioned in section 9.1.2). They help a developer focus on the task that they are currently performing. They make editing and adding new tasks possible, both recording the work to be done as well as notes on the progress or issues encountered. They allow the tasks to be prioritized based on their criticality. They support the developer's personal scheduling by helping them assign a set of tasks to be performed during a working week. This helps keep the developer on schedule and realize when they are falling behind schedule.

From a project management perspective, task-focused development allows tasks to be monitored by team leads or project managers. They can see the amount of time that has been spent on a particular task, as well as the estimate of the amount of work required to complete the task. They can also see the scheduled date of the task and use this information to ensure that the needs of the project timeline are being met.

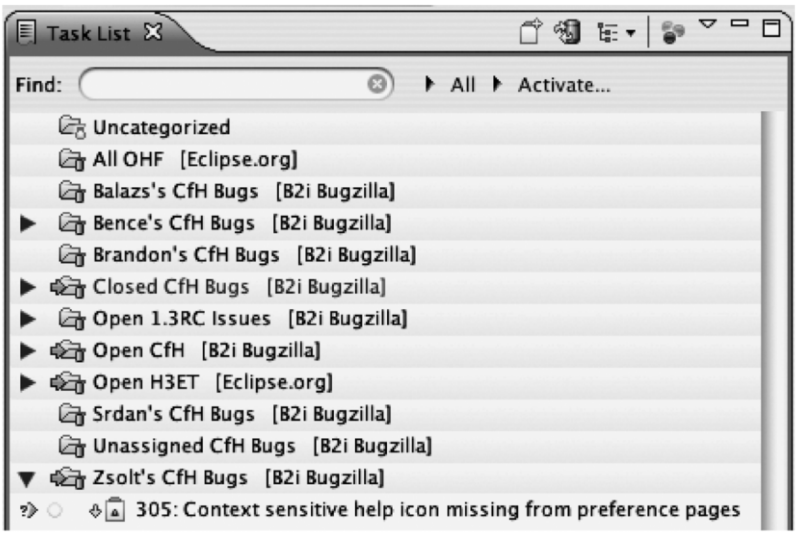
Example of task-focused development

The first feature necessary to support task-focused development is a mechanism to view, search and categorize tasks. Since the tasks may be located in several locations, both local and remote, aggregating and displaying a combined list is important.

Task categorization

Task categorization is important for both developers and managers as a simple means of viewing their tasks. To categorize the tasks, it is necessary to be able to search through the aggregated list of items for those relevant to a specific area. For example, the tasks in figure 9-10 are categorized based on their project (in this case, an Eclipse.org project called "H3ET" or "CfH") and based on the person assigned to complete the task.

Figure 9-10 Task monitoring in Eclipse with Bugzilla


Task List

Find:

All

Activate...

Uncategorized

All OHF [Eclipse.org]

Balazs's CfH Bugs [B2i Bugzilla]

Bence's CfH Bugs [B2i Bugzilla]

Brandon's CfH Bugs [B2i Bugzilla]

Closed CfH Bugs [B2i Bugzilla]

Open 1.3RC Issues [B2i Bugzilla]

Open CfH [B2i Bugzilla]

Open H3ET [Eclipse.org]

Srdan's CfH Bugs [B2i Bugzilla]

Unassigned CfH Bugs [B2i Bugzilla]

Zsolt's CfH Bugs [B2i Bugzilla]

305: Context sensitive help icon missing from preference pages

Once a category has been created, a query can be assigned to refresh and display matching tasks as shown in figure 9-11. The specific query constraints depend on the data fields of the issue tracking system. In this example, the query is restricted based on the product (NHS CfH), the component (Model Comparison Tool), the milestone (1.3RC) and the status (all pending statuses).

Since there can be dozens or hundreds of active tasks at a given time, creating several categories makes the management simple. A category can be created for the current developer, for the other members of their team, for the entire project or for high priority items. These different views make it simpler for a developer to see the tasks that they are interested in. It also simplifies the manager's task of understanding what tasks each of their team members are currently working on.

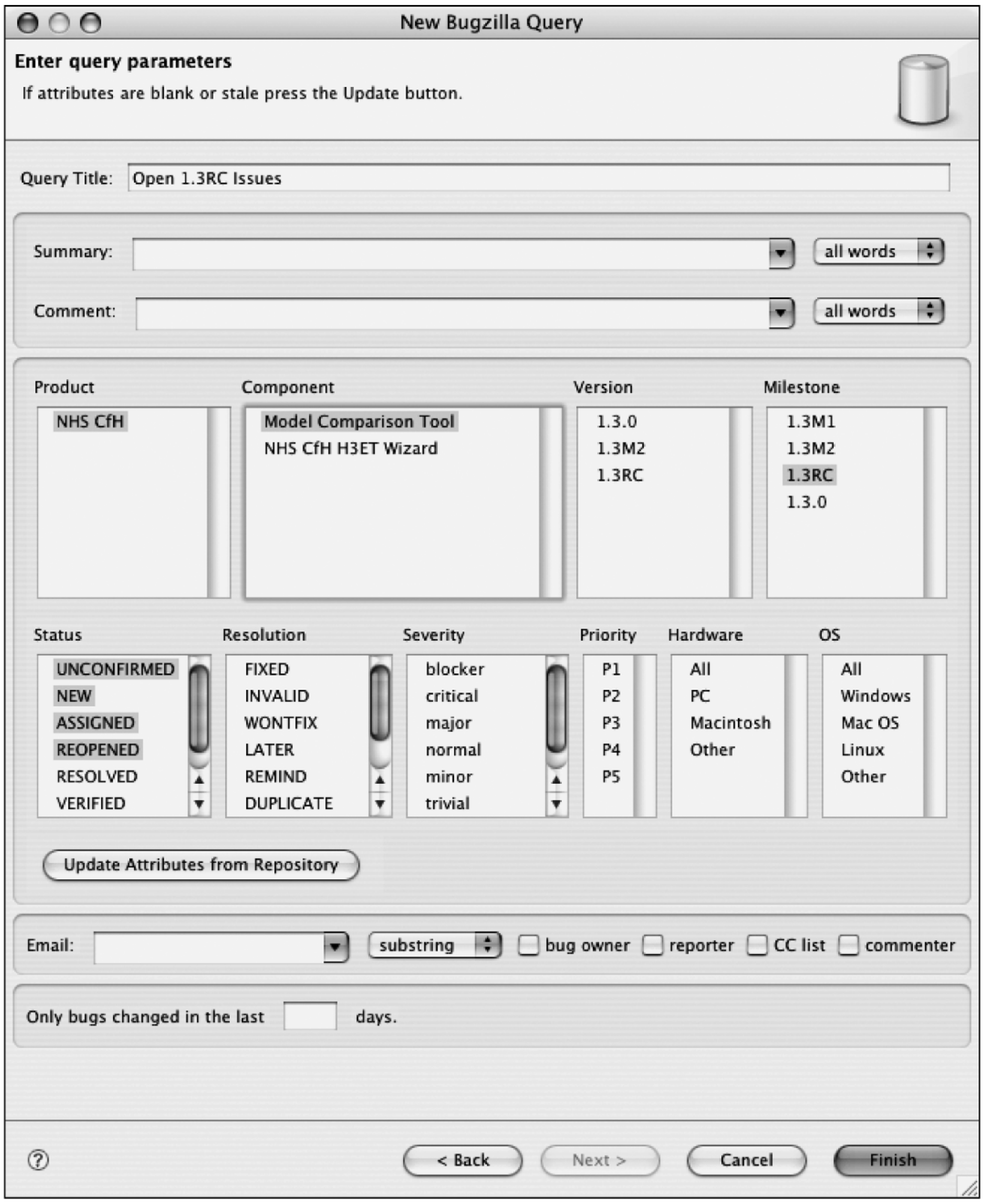
Task synchronization

Another required feature for remote task repositories is synchronization. New tasks need to be automatically added to the list, and changed or deleted tasks need to reflect their new status.

Task scheduling

Task scheduling (cf. figure 9-12) allows the developer to organize and schedule his or her working week. The effort level of the task can be estimated, such as the number of hours required for completion.

Figure 9-11 Repository query view



New Bugzilla Query

Enter query parameters

If attributes are blank or stale press the Update button.

Query Title: Open 1.3RC Issues

Summary: all words

Comment: all words

Product	Component	Version	Milestone
NHS CfH	Model Comparison Tool	1.3.0	1.3M1
	NHS CfH H3ET Wizard	1.3M2	1.3M2
		1.3RC	1.3RC
			1.3.0

Status	Resolution	Severity	Priority	Hardware	OS
UNCONFIRMED	FIXED	blocker	P1	All	All
NEW	INVALID	critical	P2	PC	Windows
ASSIGNED	WONTFIX	major	P3	Macintosh	Mac OS
REOPENED	LATER	normal	P4	Other	Linux
RESOLVED	REMIND	minor	P5		Other
VERIFIED	DUPLICATE	trivial			

Update Attributes from Repository

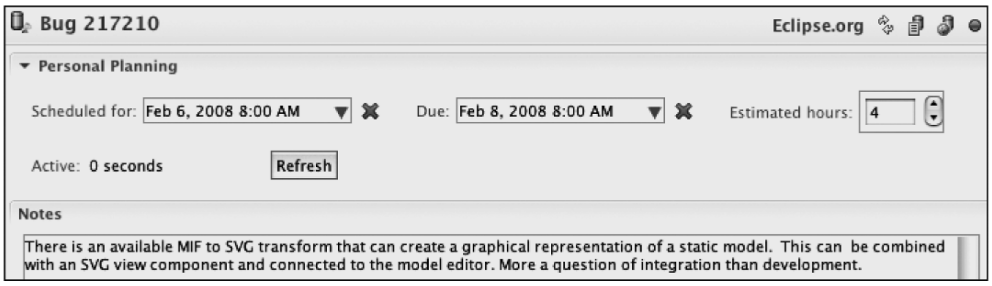
Email: substring ☐ bug owner ☐ reporter ☐ CC list ☐ commenter

Only bugs changed in the last days.

< Back Next > Cancel Finish

If a deadline exists, this can be recorded on the task. Once the developer schedules the task, they can group the tasks by those that are supposed to be completed in the coming days.

Figure 9-12 Task scheduling



Scheduled for: Feb 6, 2008 8:00 AM
Due: Feb 8, 2008 8:00 AM
Estimated hours: 4
Active: 0 seconds

There is an available MIF to SVG transform that can create a graphical representation of a static model. This can be combined with an SVG view component and connected to the model editor. More a question of integration than development.

Task scheduling helps reduce the stress associated with numerous outstanding tasks by knowing when they will be completed and when they will begin. Developers can quickly go through their list of tasks, assigning each a day of the current week, next week, or some arbitrary date in the future. This makes it quicker to order the tasks according to when they will be completed. Recording scheduling information that is shared between a team is also useful for project planning.

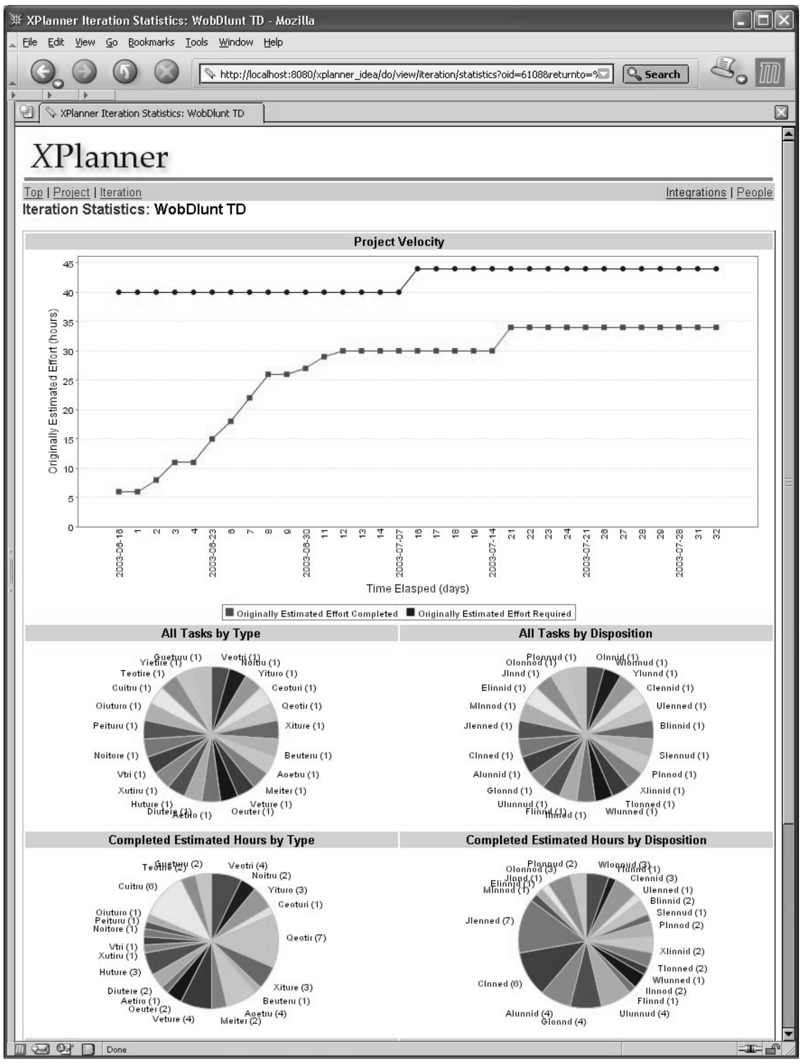
Project management support from task-focused development

The value of the task planning activities for project management support is high. Project managers can estimate the time required to complete a set of tasks, view the productivity of their employees, and reschedule tasks to ensure that a deliverable is completed in a timely manner.

Time estimated for completion

Figure 9-13 shows several graphs created with the *XPlanner* tool (http://xplanner.org). XPlanner is a project planning and tracking tool for XP teams (cf. extreme programming, section 4.4.1). The figure indicates the number of outstanding hours for a particular release. The release is defined as a collection of tasks, each of which has estimated completion values from the members of the team. Instead of manually maintaining a project list, a current overview of the amount of time estimated for completion is always available to the team.

Figure 9-13 Viewing estimated remaining hours in XPlanner[§]


XPlanner Iteration Statistics: WobDlunt TD - Mozilla
http://localhost:8080/xplanner_idea/do/view/iteration/statistics?oid=6108&returnto=%
XPlanner
Top | Project | Iteration
Integrations | People
Iteration Statistics: WobDlunt TD
Project Velocity
Originally Estimated Effort (hours)
Time Elasped (days)
Originally Estimated Effort Completed
Originally Estimated Effort Required
All Tasks by Type
All Tasks by Disposition
Completed Estimated Hours by Type
Completed Estimated Hours by Disposition
Done

§ http://xplanner.org/images/screenshots/statistics.jpg.

9.3.2 Context-based Development

The previous section highlighted some advantages of organizing work into tasks. However, the real advantages from working in a task-focused development environment accrue once the task-based development is combined with context-based development.

Reducing information overload

Context-based development is designed to reduce information overload for developers by allowing them to focus only on the programming artifacts relevant to their current task. It also allows developers to multitask much more easily, by saving the context that they were working in before they were forced to change contexts.

Task switching

A common example of this is when a high-priority task arrives that can be completed relatively quickly. The developer can save the task they are currently working on, fix the high-priority task, and then switch back to working on their original task where they left off.

Context-based tools are emerging that watch what developers do, which files they interact with, and restrict their information windows to these sets of files. This allows a developer to indicate, "I'm fixing bug #1234" and collect all the contextual information that is built up (which files were referenced, which bugs were involved) to commit to a source code control repository along with the bug fixes.

Restoring the saved context

Twelve hours or twelve days later, someone sitting across a desk or across the ocean can restore the context that was saved when fixing this problem. This decreases communication costs and increases productivity significantly, since a new developer can start off with the same set and layout of information that the original developer was working with.

When working with teams spread around the world, it is hard to lean over a colleague's shoulder and explain the changes that were recently made to one of their modules. Since development work tends to "follow the sun" on large projects, it places an increased need on communicating and collaborating with peers that are eight hours ahead or eight hours behind.

Information overload and task switching

Working with thousands of artifacts

Developers working on large systems are used to the overhead of working with thousands of artifacts distributed through dozens of modules. Separating code into modules makes it simpler for a programmer to identify the areas of the system that are affected by their changes, but it still becomes necessary to identify cross-dependencies and other relevant artifacts. Test cases, for example, are frequently located in separate modules than the code that they test; however they are immediately relevant to any work done on the code.

There is an enormous mental overhead to keeping track of each of these items, and a continual need to move between files, searching for and scrolling to a single item of interest. To cope with this, developers commonly have several editors active on different artifacts, switching between them as necessary.

Interrupted tasks

Further compounding the effect of information overload is the task switching necessary in the development environment. Some tasks of a longer and lower-priority duration will inevitably be interrupted by an urgent task affecting the entire development team. This means that the developer must switch from their current task to an entirely new task. However, during this switch, the information that had been collected about the context relevant to the currently active task is lost or forgotten.

Maintaining the relevant context

Context-based development approaches attempt to maintain the list of items relevant to the task that the developer is currently working on. This means that when the developer switches from one task to a more urgent task, they can more easily return to the same context when they finish the higher priority item.

But what artifacts are included in the context, and how is this context identified? These artifacts can be anything from products to packages, files, classes, methods and variables. Each of these items is potentially relevant to solving the developer's current task.

Attention-reactive user interfaces

The difficulty is in identifying which of the artifacts are relevant for display to the developer. Ideally, the developer's environment would display only the items that are necessary to complete the current task. This type of user interface is referred to as an *attention-reactive user interface*.

Attention-reactive user interfaces are a "general strategy for constructing interfaces for high-information applications." [Card 2002] They are composed of two parts: a *degree-of-interest (DOI) model* to describe what is interesting to the developer, and an *adaptive visual display* to optimize the visual representation of the many things to display.

9.3.3 Process-focused Automation and Tooling

Once a large number of tools are implemented and working, it becomes a challenge to tie everything together into a meaningful *process*. This is especially problematic with distributed development teams. However, between offshoring, outsourcing, and international mergers and acquisitions, distributed development teams are becoming the norm, rather than the exception.

Combining tools into a development process

Process-focused tooling tries to combine the above ideas into a customized process. For example, during a continuous integration build, a failure in one of the automated tests is identified. The specific build and test failure are linked with the bug automatically. Source code changes made to the bug and the context are also attached to it (e.g via task-focused context-based development tooling). All these links happen automatically, and the developers don't have to focus on the process; the tooling does.

Defining and enforcing the process

Furthermore, the tools help *define and enforce* a process for the distributed team members. Of course, this requires that the tooling understands the concept of a project as well as a team. When these ideas are combined, it becomes possible to assign a project to a process.

Individual processes for teams and sub-teams

Teams in a project also follow processes, so teams can have their own process defined as well. Their process may differ from the process at the project level; perhaps they are more restrictive with more rules, or less formal with fewer process rules. Generally there is a hierarchical relationship between project teams. This means that sub-teams may also refine or override the processes used by their parent groups.

Outlook: IBM Jazz

The IBM Jazz project "focuses on collaboration across geographic boundaries." [Krill 2007] It is intended to support software life cycle management by managing development artifacts throughout the development cycle [IBM 2007]. Jazz is planned to be available as commercial software in 2008.

Software process as a configurable entity

Jazz focuses on the software process as a configurable entity. The steps in the process can be defined in a workflow. Once a process has been defined, it can be connected to a project. Teams are created and assigned to the project as well. Jazz uses task-focused development (see section 9.3.1). It refers to tasks as work items, each of which are associated with a project. Since the project can be further decomposed into process steps, each of the tasks can be associated with different steps of the on-going development process. For example, a bug might be assigned to a particular milestone.

Jazz's integration with source code management systems (see section 9.1.1) extends to the work item level. Tasks can be updated with information on the changes that occurred as well as the source files that changed.

An interesting feature is the ability to suspend a current work item, such as when a high-priority bug arrives. Although task-focused development tools make it simple to switch between tasks, they do not support a simple way of saving changes to the source code that have already been made.

Suspending and resuming work items

Using Jazz, a developer can suspend a work item that is already in process. This returns their working copy of the source code to match the repository so that they can fix and commit the high-priority change. After the change has been made, they can resume their suspended work item and continue where they left off.

Process example

Eclipse development process

As an example of how process-focused development works in practice, consider a project that is being run using Jazz following the typical Eclipse software development process. This process is based on breaking a product release into frequent milestones at approximately six-week intervals [Venners 2005]. Each milestone starts with a planning phase, enters a development phase, and finishes with a stabilization step. At the end of each of the milestones is a retrospective step that evaluates the overall success of the milestone release, particularly focusing on what succeeded and what failed.

Consider how a process might be configured during the stabilization step. This step is just before the release of the milestone, and is designed to ensure the stability of the milestone release by testing and correcting as many defects as possible. Some development teams impose very high standards on any code changes during this time to ensure software stability. These often include a review process whereby any new code contributed by a team member will be peer-reviewed by another member of the team.

Violation of the process

If a developer attempted to commit the code to the repository without having their changes verified, an error would occur. This is because they violated the process. There would be steps to find out what to do next, such as a suggestion as to whom the code changes should be forwarded to in order to complete the development.

This example illustrates how process-focused development tools help to *enforce* a development process. This works across team boundaries as well, and it eases collaboration between geographically dispersed teams.

9.4 Summary and Outlook

This chapter has discussed comprehensive, integrated tool support in three primary areas that assist with global software development projects: fundamental tools, combined tools and project-wide tools.

The first set of tools provide basic features critical to distributed development projects. These tools include software version control systems, which maintain a history of a project's source code. Issue and defect tracking systems were discussed as a database of defects, enhancements and potential ways to work around existing problems. Finally, load-testing tools were outlined which can validate that programs are running as expected.

Tools that built upon these basic tools include release engineering tools, quality control tools, and continuous integration tools. Release engineering tools automate the process of software construction. Quality control tools ensure that distributed teams all work to the same standards. Continuous integration tools were described as a way to tie all of the previously discussed tools together. Continuous integration tools allow a project to be placed "on autopilot": automatically running the necessary tools whenever a project artifact changes.

Section 9.3 discussed tools that increase the productivity of distributed development teams. Task-focused development allows distributed teams to work on individually assigned issues. Context-based development allows team members to share their perspectives with colleagues regardless of their location. Process-focused tooling was discussed that helps ensure distributed teams collaborate effectively and follow a consistent process.

The needs of globally distributed development teams have grown in importance in today's globalized world. To address these needs, projects such as IBM's Jazz are hoping to address issues with distributed knowledge management and process execution. As companies attempt to optimize productivity on their global teams, the pressure to provide tooling support for worldwide team collaboration will only increase.

References

[Achour 2006] Achour, M., Betz, F., Dovgal, A. et al.: PHP Manual − Appendix A. History of PHP and Related Projects; PHP Documentation Group 2006; http://mx2.php.net/manual/en/history.php (accessed Feb 3, 2008).

[ACM 1978] Association for Computing Machinery (ACM): Proceedings of the Software Quality and Assurance Workshop, San Diego, CA, Nov 15-17, 1978; Special Joint Issue of Software Engineering Notes 3 (1978) 5 and Performance Evaluation Review 7 (1978) 3 & 4.

[Agile 2001] Agile Alliance: Manifesto for Agile Software Development; http://www.agilemanifesto.org/ (accessed Jan 9, 2008).

[Albrecht 1983] Albrecht, A., Gaftney, J.E.: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation; IEEE Transactions on Software Engineering 9 (1983) 6, pp. 639-648.

[Amberg 2005] Amberg, M., Herold, G., Kodes, R. et al.: IT Offshoring − A Cost-Oriented Analysis; in: Proceedings of Conference on Information Science, Technology and Management 2005 (CISTM 2005); New Delhi, India, July 2005.

[Ambler 2007] Ambler, S.W.: Test-Driven Development of Relational Databases; IEEE Software 26 (2007) 5, pp. 37-43.

[Ambler 2005a] Ambler, S.W.: A Manager's Introduction to The Rational Unified Process (RUP), Version December 4, 2005; http://www.ambysoft.com/downloads/managersIntroToRUP.pdf (accessed Aug 20, 2006).

[Ambler 2005b] Ambler, S.W.: The Design of a Robust Persistence Layer For Relational Databases, Version June 21, 2005; http://www.ambysoft.com/downloads/persistenceLayer.pdf (accessed May 22, 2007).

[Ambler 2003] Ambler, S.W.: Agile Database Techniques: Effective Strategies for the Agile Software Developer; Wiley Publishing, Indianapolis, IN 2003.

[Anderson 2004] Anderson, D.J.: Agile Management for Software Engineering: Applying the Theory of Constraints for Business Result; Prentice Hall, Upper Saddle River, NJ 2004.

[Andres 2006] Andres, T.: From Business Process to Application − Model-driven Development of Business Management Software; ARIS Expert Paper; IDS Scheer, Saarbrücken, Germany, May 2006.

[Apache 2008] The Apache Software Foundation: Apache JMeter User's Manual; http://jakarta.apache.org/jmeter/usermanual/ (accessed Jan 13, 2008).

[Apache 2007] The Apache Software Foundation: Apache Struts; http://struts.apache.org/ (accessed May 15, 2007).

[Apache 2006a] The Apache Software Foundation: Apache Geronimo; http://geronimo.apache.org/ (accessed Aug 10, 2006).

[Apache 2006b] The Apache Software Foundation: Apache HTTP Server Project; http://httpd.apache.org/ (accessed Aug 10, 2006).

[Apache 2006c] The Apache Software Foundation: Apache Tomcat; http://tomcat.apache.org/ (accessed May 15, 2006).

[Aspray 2006] Aspray, W., Mayadas, F., Vardi, M.Y. (Eds.): Globalization and Offshoring of Software – A Report of the ACM Job Migration Task Force; ACM 0001-0782/06/0200, Association for Computing Machinery, New York, NY 2006 (an online version is available at: http://www.acm.org/globalizationreport).

[ASPstreet 2006] ASPstreet.com: ASP Directory; http://www.aspstreet.com/directory (accessed Mar 14, 2006).

[ASUG 2006] ASUG (America's SAP User Groups): Wanted: Enterprise Architects; SAP Info (2006) 135, pp. 8-10.

[Ayers 2001] Ayers, J.B.: Handbook of Supply Chain Management; St. Lucie Press, Boca Raton, FL 2001.

[Bach 2006] Bach, J.: Rapid Software Testing; http://www.satisfice.com/info_rst.shtml (accessed Jul 14, 2007).

[Bach 2003] Bach, J.: Exploratory Testing Explained – v.1.3 4/16/03; http://www.satisfice.com/articles/et-article.pdf (accessed Jul 14, 2007).

[Bagui 2003] Bagui, S., Earp, R.: Database Design Using Entity-Relationship Diagrams; CRC Press, Portland, OR 2003.

[Baker 1972] Baker, F.T.: Chief Programmer Team of Management of Production Programming; IBM Systems Journal 11 (1972) 1, pp. 56-73.

[Barry 2003] Barry, D.: Web Services and Service-Oriented Architectures: The Savvy Manager's Guide; Morgan Kaufman Publishers, San Francisco, CA 2003.

[Basili 1975] Basili, V.R., Turner, A.J.: Iterative Enhancement: A Practical Technique for Software Development; IEEE Transactions on Software Engineering 1 (1975) 4, pp. 390-396.

[Bass 2003] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Second Edition; Addison-Wesley, Boston, MA 2003.

[Beatty 2006] Beatty, R.C., Williams, C.D.: ERP II: Best Practices for Successfully Implementing an ERP Upgrade; Communications of the ACM 49 (2006) 3, pp. 105-109.

[Beck 2007] Beck, K., Gamma, E.: JUnit Cookbook; http://junit.sourceforge.net/doc/cookbook/cookbook.htm (accessed Jul 29, 2007).

[Beck 2005] Beck, K., Andres, C.: Getting Started with XP: Toe Dipping, Racing Dives, and Cannonballs; Three Rivers Institute 2005 (available online at http://www.threeriversinstitute.org/; accessed Jan 3, 2007).

[Beck 2004] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd Edition; Addison-Wesley Professional, Boston, MA 2004.

[Becker 2007] Becker, T.: User Exits in FI/CO; http://www.sapbrainsonline.com/ARTICLES/TECHNICAL/USEREXITS/USEREXITS_in_FICO.html (accessed Aug 29, 2007).

[Bezroukov 1999] Bezroukov, N.: Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Ramondism); First Monday 4 (1999) 10.

[Bhat 2006] Bhat, J.M., Gupta, M., Murthy, S.N.: Overcoming Requirements Engineering Challenges: Lessons from Offshore Outsourcing; IEEE Software 23 (2006) 5, pp. 38-44.

[Biffl 2006] Biffl, S., Aurum, A., Boehm, B. et al.: Value-Based Software Engineering; Springer, Berlin, Heidelberg 2006.

[Boehm 1995a] Boehm, B., Clark, B., Horowitz, E. et al.: An Overview of the COCOMO 2.0 Software Cost Model; in: Proceedings of SSTC Software Technology Conference, April 1995; available for download at http://sunset.usc.edu/research/COCOMOII/cocomo_main.

[Boehm 1995b] Boehm, B., Clark, B., Horowitz, E. et al.: Cost Models for Future Software Life Cycle Processes: COCOMO 2.0; in: Arthur, J.D., Henry, S.M. (Eds.), Special Volume on Software Process and Product Measurement, Annals of Software Engineering; Baltzer Science Publishers, Amsterdam 1995; available for download at http://sunset.usc.edu/research/COCOMOII/cocomo_main. html (accessed Apr 14, 2006).

[Boehm 1981] Boehm, B.W.: Software Engineering Economics; Prentice Hall, Upper Saddle River, NJ 1981.

[Boehm 1978] Boehm, B.W., Brown, J.R. et al.: Characteristics of Software Quality; North Holland, Amsterdam, New York 1978.

[Booch 2006] Booch, G.: The Accidental Architecture; IEEE Software 23 (2006) 3, pp. 9-11.

[Booch 2005] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd Edition; Addison-Wesley Professional, Boston, MA 2005.

[Booch 1994] Booch, G.: Object-Oriented Analysis and Design with Applications, 2nd Edition; Benjamin/Cummings Publishing, Redwood City, CA 1994.

[Brooks 1995] Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition; Addison-Wesley Professional, Boston, MA 1995.

[Burn 2007] Burn, O.: Checkstyle 4.4; http://checkstyle.sourceforge.net/ (accessed Jan 12, 2008).

[Buschmann 2007] Buschmann, F., Henney, K., Schmidt, D.C.: Past, Present, and Future Trends in Software Patterns; IEEE Software 24 (2007) 4, pp. 31-37.

[Buschmann 1996] Buschmann, F., Meunier, R. et al.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns; John Wiley & Sons, New York et al. 1996.

[Campione 2001] Campione, M., Walrath, K., Huml, A.: The Java Tutorial – A Short Course on the Basics, Third Edition; Addison-Wesley, Boston, MA 2001.

[Card 2002] Card, S.K., Nation, D.: Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface; in: Proceedings of International Conference on Advanced Visual Interfaces (AVI02), Trento, Italy 2002.

[Carr 2005] Carr, N.G.: Does Software Matter? Informatik Spektrum 28 (2005) 4, pp. 271-273.

[Carr 2004] Carr, N.G.: Does IT Matter? Information Technology and the Corrosion of Competitive Advantage; Harvard Business School Publishing, Boston, MA 2004.

[Carr 2003] Carr, N.G.: IT Doesn't Matter; Harvard Business Review 81 (2003) 5, pp. 41-49.

[Cash 1992] Cash, J.I., McFarlan, F.W., McKenney, J.L.: Corporate Information Systems Management: The Issues Facing Senior Executives, 3rd Edition; Irwin Professional Publishers, Homewood, Ill. 1992.

[Chappell 2004] Chappell, D.: Enterprise Service Bus; O'Reilly Media, Sebastopol, CA 2004.

[Chen 1976] Chen, P.P.: The Entity-Relationship Model – Toward a Unified View of Data; ACM Transactions on Database Systems 1 (1976) 1, pp. 9-36.

[Clark 2004] Clark, M.: Pragmatic Project Automation; The Pragmatic Bookshelf, Raleigh, NC 2004.

[Cockburn 2004] Cockburn, A.: Crystal Clear: A Human-Powered Methodology for Small Teams; Addison-Wesley Professional, Boston, MA 2004.

[Cockburn 2000] Cockburn, A.: Writing Effective Use Cases; Addison-Wesley Professional, Boston, MA 2000.

[Cockburn 1998] Cockburn, A.: Basic Use Case Template; http://alistair.cockburn.us/images/Uctempla.doc (accessed Mar 24, 2007).

[Collins 2004] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion; O'Reilly Media, Sebastopol, CA 2004 (available online at http://svnbook.red-bean.com/ (accessed Sep 26, 2007).

[Conradi 2003] Conradi, R., Jaccheri, L., Torchiano, M.: Using software process modeling to analyze the COTS based development process; Workshop on Software Process Simulation Modeling 2003 (Prosim '03), collocated with ICSE '03, 3-4 May 2003, Portland, USA.

[Cranes 2006] Cranes Software International Ltd.: InventX – Product Overview; http://www.inventx.com/product-overview.aspx (accessed Jan 6, 2008).

[Crispin 2006] Crispin, L.: Driving Software Quality: How Test-Driven Development Impacts Software Quality; IEEE Software 25 (2006) 6, pp. 70-71.

[Cummins 2002] Cummins, F.A.: Enterprise Integration – An Architecture for Enterprise Application and Systems Integration; John Wiley & Sons, New York et al. 2002.

[Cunningham 2005] Cunningham, W.: Introduction To Fit; http://fit.c2.com/ (accessed Jul 27, 2007).

[Davies 2005a] Davies, S., Cowen, L., Giddings, C., Parker, H.: WebSphere Message Broker Basics; IBM Redbooks, White Plains, NY 2005 (updated version available online at ibm.com/redbooks; accessed Aug 31, 2007).

[Davies 2005b] Davies, S., Broadhurst, P.: WebSphere MQ V6 Fundamentals; IBM Redbooks, White Plains, NY 2005 (updated version available online at ibm.com/redbooks; accessed Aug 31, 2007).

[Davison 2003] Davison, D.: Top 10 Risks of Offshore Outsourcing; December 9, 2003; http://techupdate.zdnet.com/techupdate/stories/main/Top_10_Risks_Offshore_Outsourcing.html (accessed Mar 16, 2006).

[DCM 2006] DC&M Partners LLC: Offshore Development Process Model; http://www.dcm-partners.com/offshore-development-process-model.htm (accessed Sep 7, 2006).

[DeMarco 1978] DeMarco, T.: Structured Analysis and System Specification; Yourdon Press, New York 1978.

[Dijkstra 1970] Dijkstra, E.W.: Notes on Structured Programming, Second Edition; T.-H.-Report 70-WSK-03, Technical University Eindhoven, Netherlands 1970.

[Dijkstra 1968a] Dijkstra, E.W.: Go To Statement Considered Harmful; Communications of the ACM 11 (1968) 3, pp. 147-148.

[Dijkstra 1968b] Dijkstra, E.W.: The Structure of T.H.E. Multiprogramming System; Communications of the ACM 11 (1968) 5, pp. 341-346.

[Dittrich 2006] Dittrich, J., Mertens, P., Hau, M., Hufgard, A.: Dispositionsparameter in der Produktionsplanung mit SAP, 4. Aufl.; Vieweg, Wiesbaden, Germany 2006.

[Duvall 2007] Duvall, P.M., Matyas, S., Glover, A.: Continuous Integration, Addison-Wesley, Boston, MA 2007.

[Ebert 2005] Ebert, C., Dumke, R. et al.: Best Practices in Software Measurement – How to Use Metrics to Improve Project and Process Performance; Springer, Berlin, Heidelberg 2005.

[EBS 2006] EBS: Offshore Outsourcing Basics; http://www.ebstrategy.com/outsourcing/basics/index.htm (accessed Jan 15, 2008).

[Eclipse 2005] The Eclipse Foundation: Eclipse Process Framework Project (EPF); http://www.eclipse.org/epf/ (accessed Aug 25, 2006).

[EFC 2006] European Foundation Centre (EFC): Funders Online; http://www.fundersonline.org/grantseekers/proposal_basics.html (accessed Jan 14, 2008).

[Elmasri 2006] Elmasri, R., Shamkant B., Navathe, S.B.: Fundamentals of Database Systems, 5th Edition; Addison-Wesley, Boston, MA 2006.

[Endrei 2004] Endrei, M., Ang, J., Arsanjani, A. et al.: Patterns: Service-Oriented Architecture and Web Services; IBM Redbooks, White Plains, NY 2004 (available online at ibm.com/redbooks; accessed Apr 23, 2006).

[Eriksson 2004] Eriksson, H.-E., Penker, M. et al.: UML 2 Toolkit; John Wiley & Sons, New York et al. 2003.

[Finnegan 2007] Finnegan, D., Willcocks, L.P.: Implementing CRM: From Technology to Knowledge; John Wiley & Sons, New York et al. 2007.

[Fitzgerald 2006] Fitzgerald, B.: The Transformation of Open Source Software; MIS Quarterly 30 (2006) 3, pp. 587-598.

[Fleming 2006] Fleming, Q.W., Koppelman, J.M.: Earned Value Project Management, 3rd Edition; Project Management Institute, Newton Square, PA 2006.

[Fowler 2006] Fowler, M.: Continuous Integration – last updated May 1, 2006; http://martinfowler.com/articles/continuousIntegration.html (accessed Jan 16, 2008).

[Gamma 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley Professional, Boston, MA 1995.

[Gane 1979] Gane, C., Sarson, T.: Structured Systems Analysis: Tools and Techniques; Prentice Hall, Englewood Cliffs, NJ 1979.

[Greer 2006] Greer, M.: What's Project Portfolio Management (PPM) & Why Should Project Managers Care About It?; http://www.michaelgreer.com/ppm.htm (accessed Jan 9, 2008).

[Gulzar 2002] Gulzar, N.: Fast Track to Struts: What it Does and How; The ServerSide.com, Nov 4, 2002 (available online at http://www.theserverside.com/tt/articles/content/StrutsFastTrack/StrutsFastTrack.pdf; accessed Apr 23, 2007).

[Gurses 2005] Gurses, L.: Improving Code Quality with PMD and Eclipse; EclipseZone.com October 2005; http://www.eclipsezone.com/articles/pmd/ (accessed Jan 16, 2008).

[Highsmith 2001] Highsmith, J.: History: The Agile Manifesto; http://www.agilemanifesto.org/history.html (accessed Jan 9, 2008).

[Hillside 2007] Hillside.net: The Pattern Abstracts from Pattern-Oriented Software Architecture; http://hillside.net/patterns/books/Siemens/abstracts.html (accessed April 10, 2007).

[Hoffer 2006] Hoffer, J.A., McFadden, F., Prescott, M.: Modern Database Management; Pearson Education, Upper Saddle River, NJ 2006.

[Hohpe 2004] Hohpe, G., Woolf, B.: Enterprise Integration Patterns – Designing, Building, and Deploying Messaging Solutions; Addison-Wesley, Boston 2004.

[Homer 2007] Homer Computer Services: Advanced Planning System Software Selection Guide; http://www.homercomputer.com.au/pdf/wapssamples.pdf (accessed Aug 16, 2007).

[Huang 2006] Huang, L., Boehm, B.: How Much Software Quality Investment Is Enough: A Value-Based Approach; IEEE Software 23 (2006) 5, pp. 88-95.

[Hunt 1999] Hunt, A., Thomas, D,: The Pragmatic Programmer; Addison-Wesley, Boston 1999.

[IBM 2008] IBM Corp.: WebSphere Software; http://www-306.ibm.com/software/websphere/ (accessed Jan 3, 2008).

[IBM 2007] IBM Corp.: All that Jazz; http://www.alphaworks.ibm.com/topics/cde (accessed Jan 13, 2008).

[IBM 2003] IBM Corp.: WebSphere MQ Application Programming Guide, Fourth Edition; IBM Corp., White Plains, NY 2003.

[IEEE 1998a] Institute of Electrical and Electronics Engineers, Inc. (IEEE): IEEE Standard for Software Test Documentation, IEEE 829-1998; New York, NY 1998.

[IEEE 1998b] Institute of Electrical and Electronics Engineers, Inc. (IEEE): Standard for Software Reviews, IEEE 1028-1997; New York, NY 1998.

[IFPUG 2005] International Function Point Users Group (IFPUG): Publications and Products; Nov. 2005; http://www.ifpug.org/publications (accessed Dec 1, 2007).

[Incava 2008] Incava.org: DoctorJ; http://www.incava.org/projects/java/doctorj/ (accessed Jan 12, 2008).

[Infosys 2008] Infosys Technologies Ltd.: Global Delivery Model – Risk Mitigation; http://www.infosys.com/global-sourcing/global-delivery-model/risk-mitigation.asp (accessed Jan 17, 2008).

[Infosys 2007] Infosys Technologies Ltd.: Investor Services – Investor FAQs – Corporate Information; http://www.infosys.com/investors/investor-services/FAQs.asp (accessed Jan 1, 2008).

[IPMA 2006] International Project Management Association (IPMA): ICB – IPMA Competence Baseline (ICB), Version 3.0; IPMA, Nijkerk, Netherlands 2006.

[Jacobson 1995] Jacobson, I., Ericsson, M., Jacobson, A.: The Object Advantage: Business Process Reengineering With Object Technology; ACM Press, New York, NY 1995.

[Jankowska 2005] Jankowska, A.M., Kurbel, K.: Service-Oriented Architecture Supporting Mobile Access to an ERP System; in: Ferstl, O. et al. (Eds.), Wirtschaftsinformatik 2005 – eEconomy, eGovernment, eSociety; Physica, Heidelberg, Germany 2005; pp. 371-390.

[Jeffries 2007] Jeffries, R., Melnik, G.: TDD: The Art of Fearless Programming; IEEE Software 26 (2007) 5, pp. 24-30.

[Jendrock 2007] Jendrock, E., Ball, J., Carlson, D. et al.: The Java EE 5 Tutorial; http://java.sun.com/javaee/5/docs/tutorial/doc/; Sun Microsystems 2007 (accessed Dec 8, 2007).

[Johnson 2007] Johnson, K.: Open-Source Software Development; Los Angeles Chinese Learning Center; http://chinese-school.netfirms.com/computer-article-open-source.html (accessed Feb 17, 2007).

[Johnson 2001] Johnson, K.: A Descriptive Process Model for Open-Source Software Development; Master's Thesis, University of Calgary, Department of Computer Science, June 2001.

[Jupitermedia 2005] Jupitermedia Corp.: Understanding LAMP; December 1, 2005; http://www.serverwatch.com/tutorials/article.php/3567741 (accessed Jul 24, 2006).

[Juristo 2006] Juristo, N., Moreno, A.M., Vegas, S., Solari, M.: In Search of What We Experimentally Know about Unit Testing; IEEE Software 25 (2006) 6, pp. 72-79.

[Kagermann 2006] Kagermann, H., Österle, H.: Geschäftsmodelle 2010 – Wie CEOs Unternehmen transformieren; FAZ Buch, Frankfurt/Main, Germany 2006.

[Kan 2002] Kan, S.H.: Metrics and Models in Software Quality Engineering, 2nd Edition; Addison-Wesley Longman, Amsterdam, Netherlands 2002.

[Kendall 2005] Kendall, K.E., Kendall, J.E.: Systems Analysis and Design, Sixth Edition; Prentice Hall, Upper Saddle River, NJ 2005.

[Kircher 2007] Kircher, M., Völter, M.: Software Patterns; IEEE Software 24 (2007) 4, pp. 28-30.

[Kleppe 2003] Kleppe, A., Warmer, J., Bast, T.: MDA Explained: The Model Driven Architecture – Practice and Promise; Addison-Wesley, Boston, MA 2003.

[Krill 2007] Krill, P.: IBM sings Jazz tune for app development, Jun 13, 2007; http://www.arnnet.com.au/index.php/id;174308548;fp;16;fpid;1 (accessed Jan 24, 2008).

[Kroll 2005] Kroll, P.: Introducing IBM Rational Method Composer; http://www-128.ibm.com/developerworks/rational/library/nov05/kroll/ (accessed Aug 25, 2006).

[Kruchten 2006] Kruchten, P., Obbink, H., Stafford, J.: The Past, Present, and Future of Software Architecture; IEEE Software 23 (2006) 2, pp. 22-30.

[Kruchten 1996] Kruchten, P.: A Rational Development Process; Crosstalk 9 (1996) 7, pp. 11-16.

[Kunze 1998] Kunze, M.: Lasst es leuchten; LAMP: Datenbankgestütztes Web-Publishing-System mit Freeware; c't (1998) 12, p. 230.

[Kurbel 2006] Kurbel, K., Schreber, D., Ulrich, B.: A Web Services Façade for an Open Source ERP System; in: Proceedings of the 12th Americas Conference on Information Systems, Acapulco, Mexico, August 4-6, 2006.

[Kurbel 2005] Kurbel, K.: Enterprise Resource Planning and Integration; in: Khosrow-Pour, M. (Ed.): Encyclopedia of Information Science and Technology, Vol. I-V; Idea Group Reference, Hershey, PA et al. 2005; pp. 1075-1082.

[Kurbel 1992] Kurbel, K., Dornhoff, P.: A System for Case-Based Effort Estimation for Software-Development Projects; Working Paper No. 11 of the Institute of Business Informatics, University of Muenster, Germany, July 1992.

[Kurbel 1990] Kurbel, K.: An Integrated Approach to Expert System Development, Project Organization, and Project Management; in: SE90 – Proceedings of Software Engineering 90, Brighton, July 1990; Cambridge 1990, pp. 271-281.

[Kurbel 1987] Kurbel, K., Labentz, M., Pietsch, W.: Prototyping und Projektmanagement bei großen Entwicklungsteams; Information Management 2 (1987) 1, S. 6-15.

[Larmann 2005] Larmann, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition; Prentice Hall Ptr, Upper Saddle River, NJ 2005.

[Laudon 2007] Laudon, K.C., Laudon, J.P.: Management Information Systems: Managing the Digital Firm, Tenth Edition; Prentice Hall, Upper Saddle River, NJ 2007.

[Manolescu 2007] The Growing Divide in the Patterns World; IEEE Software 24 (2007) 4, pp. 61-67.

[Martin 2007] Martin, R.C.: Professionalism and Test-Driven Development; IEEE Software 26 (2007) 5, pp. 32-36.

[Martin 2003] Martin, R.C.: Agile Software Development − Principles, Patterns, and Practices; Prentice Hall, Upper Saddle River, NJ 2003.

[Martin 1990] Martin, J.: Information Engineering, Book II: Planning and Analysis; Prentice Hall, Englewood Cliffs, NJ 1990.

[Martin 1989] Martin, J.: Information Engineering, Book I: Introduction; Prentice Hall, Englewood Cliffs, NJ 1989.

[Martin 1986] Martin, J., McClure, C.: Structured Techniques – The Basis for CASE; Prentice Hall, Englewood Cliffs, NJ 1990.

[Mason 2006] Mason, M.: Pragmatic Version Control Using Subversion, 2nd Edition; The Pragmatic Bookshelf, Raleigh, NC 2006.

[McDowell 2006] McDowell, C., Werner, L., Bullock, H.E., Fernald, J.: Pair Programming Improves Student Retention, Confidence, and Program Quality; Communications of the ACM 49 (2006) 8, pp. 90-95.

[Meyr 2002] Meyr, H., Wagner, M., Rohde, J.: Structure of Advanced Planning Systems; in: Stadtler, H., Kilger, C. (Eds.), Supply Chain Management and Advanced Planning, 2nd Edition; Springer, New York 2002, pp. 99-104.

[Microsoft 2008a] Microsoft Corp.: Microsoft Office Project Portfolio Server 2007; http://office.microsoft.com/en-us/portfolioserver/FX101674151033.aspx (accessed Jan 7, 2008).

[Microsoft 2008b] Microsoft Corp.: Microsoft Office Project Server 2007; http://office.microsoft.com/en-us/projectserver/FX100739841033.aspx (accessed Jan 7, 2008).

[Microsoft 2007a] Microsoft Corp.: Microsoft Server Products Overview; http://www.microsoft.com/servers/overview.mspx (accessed Dec 7, 2007).

[Microsoft 2007b] Microsoft Corp.: .NET Framework Developer Center − Technology Overview; http://msdn2.microsoft.com/en-us/netframework/ (accessed Dec 7, 2007).

[Microsoft 2007c] Microsoft Corp.: .NET Framework Developer's Guide − .NET Framework Conceptual Overview; http://msdn2.microsoft.com/en-us/library/ (accessed Dec 7, 2007).

[Mills 1980] Mills, H.D.: The Management of Software Engineering, Part I: Principles of Software Engineering; IBM Systems Journal 19 (1980) 4, pp. 415-419.

[Mills 1973] Harlan D., Mills, H.D., Baker, F.T.: Chief Programmer Teams; Datamation 19 (1973) 2, pp. 58-61.

[Morisio 2002] Morisio, M., Seaman, C.B., Basili, V.R., et al.: COTS-Based Software Development: Processes and Open Issues; Journal of Systems and Software 61 (2002) 3, pp. 189-199.

[Morisio 2000] Morisio, M., Seaman, C.B., Parra, A.T. et al.: Investigating and Improving a COTS-Based Software Development Process; in: Proceedings of the International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 4-11, 2000; IEEE Computer Society Press, Los Alamitos, CA, pp. 32-41.

[Morrison 2005] Morrison, P., Macia, M.: Offshoring: All Your Questions Answered; http://www.alsbridge.com/outsourcing_leadership/dec2005_offshoring.shtml (accessed Jan 16, 2008).

[Mozilla 2007] Mozilla Foundation: The Bugzilla Guide – 3.1.2 Development Release; http://www.bugzilla.org/docs/ (accessed Jan 13, 2008).

[Myers 1976] Myers, G.J.: Software Reliability – Principles and Practices; John Wiley & Sons, New York et al. 1976.

[Newcomer 2004] Newcomer, E., Lomow, G.: Understanding SOA with Web Services; Addison-Wesley Professional, Boston, MA 2004.

[Nilekani 2006] Nilekani, M.: IT, India, the Zenith; Sunday Hindustan Times (2006) January 1, p. 5.

[NTT 2006] NTT DoCoMo, Inc.: i-mode; http://www.nttdocomo.com/corebiz/services/imode/index.html (accessed Jan 2, 2006).

[Nuseibeh 2000] Nuseibeh, B.A., Easterbrook, M.A.: Requirements Engineering: A Roadmap; in: Finkelstein, A.C.W. (Ed.), Proceedings of the Conference on The Future of Software Engineering; Limerick, Ireland, June 4-11, 2000; IEEE Computer Society Press, Los Alamitos, CA, pp. 35-46.

[OASIS 2006] OASIS Open, Inc.: Web Services Business Process Execution Language Version 2.0, Committee Draft, 17 May, 2006; http://www.oasis-open.org/committees/download.php/18714/wsbpel-specification-draft-May17.htm (accessed Feb 16, 2007).

[Offshore 2008] OffshoreXperts.com: Outsourcing Services Directory; http://www.offshorexperts.com (accessed Jan 6, 2008).

[OMG 2007] The Object Management Group (OMG): CORBA Basics; http://www.omg.org/gettingstarted/corbafaq.htm (accessed Jan 5, 2008).

[OMG 2006] The Object Management Group (OMG): OMG Model Driven Architecture; http://www.omg.org/mda/ (accessed Aug 16, 2006).

[Oracle 2006] Oracle Corp.: Anatomy of an XML Database: Oracle Berkeley DB XML – An Oracle White Paper; Redwood Shores, CA, September 2006; http://www.oracle.com/dm/07h1corp/bdbxml.pdf (accessed May 22, 2007).

[Overby 2003] Overby, S.: The Hidden Costs of Offshore Outsourcing; CIO Magazine (2003) Sep 1; online available at http://www.cio.com/archive/090103/money.html (accessed Feb 2, 2008).

[Parnas 1979] Parnas, D.L.: Designing Software for Ease of Extension and Contraction; IEEE Transactions on Software Engineering 5 (1979) 2, pp. 128-138.

[Parnas 1974] Parnas, D.L.: On a 'Buzzword': Hierarchical Structure; in: Rosenfeld, J.L. (Ed.), Information Processing 74, Proceedings of IFIP Congress 74; Amsterdam, London 1974, pp. 339-344.

[Parnas 1972a] Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules; Communications of the ACM 15 (1972) 12, pp. 1053-1058.

[Parnas 1972b] Parnas, D.L.: Information Distribution Aspects of Design Methodology; in: Freiman, C.V. (Ed.), Information Processing 71, Proceedings of IFIP Congress 71, Volume 1 − Foundations and Systems; Amsterdam, London 1972, pp. 339-344.

[Paulish 2007] Paulish, D.: Methods, Processes & Tools for Global Software Development; Presentation at the International Workshop on Tool Support and Requirements Management in Distributed Projects (REMIDI'07), Munich, August 27-30, 2007; http://www4.in.tum.de/~kuhrmann/remidi07.shtml (accessed Dec 24, 2007).

[Perks 2003] Perks, M.: Guide to Running Software Development Projects; http://www-128.ibm.com/developerworks/websphere/library/techarticles/0306_perks/ perks.html (accessed Jan 24, 2008).

[Planisware 2007] Planisware USA: Project Portfolio Management & Strategic Planning; http://www.planisware.com/generique.php?docid=28 (accessed Jan 1, 2008).

[PM 2007] PM-Software.info: Projektmanagement-Software; http://www.pm-software.info/produktliste.html (accessed Jan 4, 2008).

[PMI 2004] Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK Guide), Third Edition; Project Management Institute, Newton Square, PA 2004.

[Rational 1998] Rational Software Corp.: Rational Unified Process − Best Practices for Software Development Teams, White Paper; Rational Software Corp., Cupertino, CA 1998.

[Raymond 2000] Raymond. E.S.: The Cathedral and the Bazaar, Version 3.0, September 2000; http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/cathedral-bazaar.ps (accessed Feb 14, 2007).

[Richardson 2005] Richardson, J.R., Gwaltney, W.A. Jr.: Ship It! A Practical Guide to Successful Software Projects; The Pragmatic Bookshelf, Raleigh, NC 2005.

[Ross 1977] Ross, D., Schomann, K.: Structured Analysis for Requirements Definition; IEEE Transactions on Software Engineering SE-3 (1977) 1, pp. 6-15.

[Rothman 2005] Rothman, J., Derby, E.: Behind Closed Doors: Secrets of Great Management; The Pragmatic Bookshelf, Raleigh, NC 2005.

[Rothman 1997] Rothman, J.: Iterative Software Project Planning and Tracking; http://www.jrothman.com/Papers/7ICSQ97.html (accessed Jan 14, 2008).

[Royce 1970] Royce, W.W.: Managing the Development of Large Software Systems: Concepts and Techniques; IEEE WESCON Technical Papers, Western Electronic Show and Convention, Los Angeles, Aug. 25-28, 1970, pp. 1-9; reprinted in: Riddle, W.E. (Ed.), Proceedings of the 9th International Conference on Software Engineering, Monterey, CA; IEEE Computer Society Press, Los Alamitos, CA, March 1987, pp. 328-338.

[Ruh 2001] Ruh, W.A., Maginnis, F.X., Brown, W.J.: Enterprise Application Integration: A Wiley Tech Brief; John Wiley & Sons, New York et al. 2001.

[Ruiz 2007] Ruiz, A., Price, Y.W.: Test-Driven GUI Development with TestNG and Abbot; IEEE Software 26 (2007) 5, pp. 51-57.

[Sadtler 2005] Sadtler, C., Laursen, L.B., Phillips, M. et al.: WebSphere Application Server V6 System Management and Configuration Handbook; IBM Redbooks, White Plains, NY 2005 (available online at http://ibm.com/redbooks; accessed Jan 31, 2008).

[Sagawa 1990] Sagawa, J.M.: Repository Manager technology; IBM Systems Journal 29 (1990) 2, pp. 209-227.

[Sakthivel 2007] Sakthivel, S.: Managing Risk in Offshore Systems Development; Communications of the ACM 50 (2007) 4, pp. 69-75.

[Salesforce 2006] Salesforce.com; http://www.salesforce.com (accessed Mar 14, 2006).

[SAP 2008] SAP AG: Components and Tools of SAP NetWeaver: SAP Solution Manager; http://www.sap.com/platform/netweaver/components/solutionmanager/index.epx (accessed Jan 1, 2008).

[SAP 2007a] SAP AG: Enterprise Service-oriented Architecture – Design, Development, and Deployment – SAP Solution Brief, SAP NetWeaver; SAP AG, Walldorf, Germany 2007.

[SAP 2007b] SAP AG: SAP Composite Application Framework: A Robust Environment for the Design of Composite Applications; SAP AG, Walldorf, Germany 2007; http://www.sap.com/platform/netweaver/cafindex.epx (accessed May 17, 2007).

[SAP 2007c] SAP AG: SAP ERP – SAP Solution Map; http://www.sap.com/solutions/business-suite/erp/index.epx (accessed May 25, 2007).

[SAP 2006a] SAP AG: IT Practices with SAP NetWeaver: The Best Solutions for your Business Requirements – SAP Solution in Detail, SAP NetWeaver; SAP AG, Walldorf, Germany 2006.

[SAP 2006b] SAP AG: SAP NetWeaver and Enterprise Services Architecture – SAP Solution in Detail, SAP NetWeaver; SAP AG, Walldorf, Germany 2006.

[SAP 2005a] SAP AG: Enterprise Services Architecture – Design, Development, and Deployment – SAP Solution Brief, SAP NetWeaver; SAP AG, Walldorf, Germany 2005.

[SAP 2005b] SAP AG: mySAP Supply Chain Management – Solution Overview; SAP AG, Walldorf, Germany 2005.

[SAP 2005c] SAP AG: R/3 System – SAP Knowledge Management; SAP Documentation Products and Services; SAP AG, Walldorf 2005.

[SAP 2004a] SAP AG: Enterprise Services Architecture − An Introduction − SAP White Paper, SAP NetWeaver; SAP AG, Walldorf, Germany 2004.

[SAP 2004b] SAP AG: SAP NetWeaver; SAP AG, Walldorf, Germany 2004.

[SAP 2003] SAP AG: SAP NetWeaver Platform Interoperability with IBM WebSphere and Microsoft .NET − SAP White Paper, SAP NetWeaver; SAP AG, Walldorf, Germany 2003.

[SAP 1997] SAP AG: R/3 System – SAP Technology Infrastructure – From Client/Server to Internet Architecture; SAP AG, Walldorf, Germany 1997.

[SCC 2006] Supply-Chain Council: Supply-Chain Operations Reference-model − Version 8.0; Washington, DC 2006.

[Scheer 2005] Scheer, A.-W., Thomas, O., Adam, O.: Process Modeling Using Event-driven Process Chains. In: Dumas, M. et al. (Eds.), Process-Aware Information Systems – Bridging People and Software through Process Technology; John Wiley & Sons, Hoboken, NJ 2005, pp. 119-145.

[Scheer 2002] Scheer, A.-W., Abolhassan, F. et al.: Business Process Excellence − ARIS in Practice; Springer, Berlin, Germany 2002.

[Scheer 2000] Scheer, A.-W.: ARIS − Business Process Modeling, 3rd Edition; Springer, Berlin, Germany 2000.

[Scherer 2004] Scherer, E.: ERP-Zufriedenheit: Best Practises und Best Fit bei der ERP-Systemauswahl; PPS Management 9 (2004) 2, pp. 38-40.

[SEI 2007] Software Engineering Institute (SEI): Capability Maturity Model Integration (CMMI) Version 1.2 Overview; http://www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf (accessed Dec 15, 2007).

[Serrano 2006] Serrano, N., Sarriegi, J.M.: Open Source Software ERPs: A New Alternative for an Old Need; IEEE Software 23 (2006) 3, pp. 94-97.

[Shaw 2006] Shaw, M., Clements, P.: The Golden Age of Software Architecture; IEEE Software 23 (2006) 2, pp. 31-39.

[Shneiderman 2005] Shneiderman, B., Plaisant, C.: Designing the User Interface: Strategies for Effective Human-Computer Interaction, 4th Edition; Addison-Wesley, Boston, MA 2005.

[Siebel 2006] Siebel Systems, Inc.: What is CRM?; http://www.siebel.com/what-is-crm/software-solutions.shtm (accessed Jan 2, 2006).

[Singh 2002] Singh, I., Stearns, B. et al.: Designing Enterprise Applications with the J2EE Platform, Second Edition; Addison-Wesley Professional, Boston, MA 2002.

[Sommerville 2007] Sommerville, I.: Software Engineering, Eighth Edition; Addison-Wesley, Harlow, UK 2007.

[Standish 2004] The Standish Group International, Inc.: 2004 Third Quarter Research Report; http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf (accessed Jan 8, 2006).

[Stevens 1974] Stevens, W., Myers, G., Constantine, L.: Structured Design; IBM Systems Journal 13 (1974) 2, pp. 115-139.

[Stone 1993] Stone, J.A.: Inside ADW and IEF − The Promise and the Reality of CASE; McGraw-Hill, New York 1993.

[Subramaniam 2006] Subramaniam, V., Hunt, A.: Practices of an Agile Developer; The Pragmatic Bookshelf, Raleigh, NC 2006.

[Sun 2007] Sun Microsystems, Inc.: Reference API Specifications; http://java.sun.com/reference/api/ (accessed July 1, 2007).

[Sun 2006a] Sun Microsystems, Inc.: Java Pet Store Demo; https://blueprints.dev.java.net/petstore/; Sun Microsystems 2006 (accessed April 15, 2007).

[Sun 2006b] Sun Microsystems, Inc.: JDK 6 Documentation; http://java.sun.com/javase/6/docs/Sun Microsystems 2006 (accessed Dec 2, 2007).

[Sun 2004] Sun Microsystems, Inc.: Java™ Remote Method Invocation (Java RMI); http://java.sun.com/j2se/1.3/docs/guide/rmi/ (accessed May 15, 2006).

[Sun 2002] Sun Microsystems, Inc.: Java BluePrints – Model-View-Controller; http://java.sun.com/blueprints/patterns/MVC-detailed.html (accessed April 8, 2007).

[Sun 1999] Sun Microsystems, Inc.: Code Conventions for the Java Programming Language – Revised April 20, 1999; http://java.sun.com/docs/codeconv/ (accessed Jan 12, 2008).

[Takagiwa 2002] Takagiwa, O., Korchmar, J., Lindquist, A., Vojtko, M.: WebSphere Studio Application Developer Programming Guide; IBM Redbooks, San Jose, CA 2002 (available online at http://www.redbooks.ibm.com/; accessed Jan 23, 2008).

[Tatvasoft 2006a] Tatvasoft: Software Outsourcing Issues; http://www.tatvasoft.com/software-outsourcing-central/outsourcing_issues.asp (accessed Mar 15, 2006).

[Tatvasoft 2006b] Tatvasoft: Software Outsourcing Process; http://www.tatvasoft.com/software-outsourcing-central/development/default.asp (accessed Mar 15, 2006).

[Tatvasoft 2006c] Tatvasoft: Software Outsourcing − Tatvasoft Advantages; http://www.tatvasoft.com/software-outsourcing/software-outsourcing-tatva .asp (accessed Mar 15, 2006).

[TCS 2007] Tata Consultancy Services (TCS): Results for Quarter II FY 2007-2008; http://www.tcs.com/Investors/pdf/TCS_Analysts_Q2_08.pdf (accessed Jan 1, 2008).

[van der Lans 2006] van der Lans, R.F.: Introduction to SQL – Mastering the Relational Database Language, 4th Edition; Addison-Wesley Professional, Boston, MA 2006.

[Venners 2005] Venners, B.: Eclipse's Culture of Shipping, June 28, 2005; http://www.artima.com/lejava/articles/eclipse_culture.html (accessed Jan 15, 2008).

[W3C 2006] World Wide Web Consortium (W3C): Web Content Accessibility Guidelines 2.0 – W3C Working Draft 27 April 2006; http://www.w3.org/TR/WCAG20/ (accessed Mar 15, 2007).

[W3C 2004] World Wide Web Consortium (W3C): Web Services Glossary; http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/ (accessed Apr 22, 2006).

[W3C 2003] World Wide Web Consortium (W3C): SOAP Version 1.2 Part 1: Messaging Framework – W3C Recommendation 24 June 2003; http://www.w3.org/TR/soap12-part1/ (accessed Apr 22, 2006).

[W3C 2001] World Wide Web Consortium (W3C): Web Services Description Language (WSDL) 1.1 – W3C Note 15 March 2001; http://www.w3.org/TR/wsdl (accessed Apr 23, 2006).

[W3C 1999] World Wide Web Consortium (W3C): Web Content Accessibility Guidelines 1.0; W3C Recommendation 5-May-1999; http://www.w3.org/TR/WCAG10/ (accessed Jan 3, 2008).

[Wells 2006] Wells, D.: Extreme Programming: A Gentle Introduction; http://www.extremeprogramming.org/ (accessed Feb 1, 2008).

[West 2003] West, D.: Planning a Project with the IBM Rational Unified Process; IBM Corp., Somers, NY 2003 (available online at http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/tp151.pdf; accessed Aug 21, 2006).

[Wiegers 2003] Wiegers, K.E.: So You Want To Be a Requirements Analyst? Software Development 11 (2003) 7 (available online at http://www.process-impact.com/articles/be_analyst. pdf; accessed Mar 24, 2007).

[Wiegers 2001] Wiegers, K.E.: Requirements When the Field Isn't Green; STQE 3 (2001) 3 (available online at http://www.processimpact.com/articles/reqs_not_green.pdf; (accessed Mar 23, 2007).

[Woods 2006] Woods, D.: Packaged Composite Applications: A Liberating Force for the User Interface; SAP Design Guild Website, http://www.sapdesign-guild.org/editions/edition7/composite_applications.asp (accessed Jan 8, 2006).

[Wyomissing 2008] Wyomissing Publishing: PM Digest – Project Management Software; http://www.pmdigest.com/software/ (accessed Jan 2, 2008).

[XML:DB 2003] XML:DB Initiative for XML Databases: What is an XML database?; http://xmldb-org.sourceforge.net/faqs.html (accessed May 23, 2007).

[Yuan 2007] Yuan, M.J., Heute, T.: JBoss Seam: Simplicity and Power Beyond Java EE; Prentice Hall PTR, Indianapolis, IN 2007.

[Yourdon 1989] Yourdon, E.: Modern Structured Analysis; Yourdon Press, Englewood Cliffs, NJ 1989.

[Yourdon 1979] Yourdon, E., Constantine, L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design; Yourdon Press, Englewood Cliffs, NJ 1979.

[Zachman 1982] Zachman, J.A.: Business Systems Planning and Business Information Control Study: A comparison; IBM Systems Journal 21 (1982) 1, pp. 31-53.

[Zakhour 2006] Zakhour, S., Hommel, S. et al.: The Java Tutorial: A Short Course on the Basics, 4th Edition; Prentice Hall PTR, Indianapolis, IN 2006.

Index