



1

Introduction

Objectives

The objectives of this chapter are to introduce software engineering and to provide a framework for understanding the rest of the book. When you have read this chapter you will:

- understand what software engineering is and why it is important;
- understand that the development of different types of software systems may require different software engineering techniques;
- understand some ethical and professional issues that are important for software engineers;
- have been introduced to three systems, of different types, that will be used as examples throughout the book.

Contents

- 1.1** Professional software development
- 1.2** Software engineering ethics
- 1.3** Case studies

We can't run the modern world without software. National infrastructures and utilities are controlled by computer-based systems and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Entertainment, including the music industry, computer games, and film and television, is software intensive. Therefore, software engineering is essential for the functioning of national and international societies.

Software systems are abstract and intangible. They are not constrained by the properties of materials, governed by physical laws, or by manufacturing processes. This simplifies software engineering, as there are no natural limits to the potential of software. However, because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change.

There are many different types of software systems, from simple embedded systems to complex, worldwide information systems. It is pointless to look for universal notations, methods, or techniques for software engineering because different types of software require different approaches. Developing an organizational information system is completely different from developing a controller for a scientific instrument. Neither of these systems has much in common with a graphics-intensive computer game. All of these applications need software engineering; they do not all need the same software engineering techniques.

There are still many reports of software projects going wrong and 'software failures'. Software engineering is criticized as inadequate for modern software development. However, in my view, many of these so-called software failures are a consequence of two factors:

1. *Increasing demands* As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible. Existing software engineering methods cannot cope and new software engineering techniques have to be developed to meet new these new demands.
2. *Low expectations* It is relatively easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be. We need better software engineering education and training to address this problem.

Software engineers can be rightly proud of their achievements. Of course we still have problems developing complex software but, without software engineering, we would not have explored space, would not have the Internet or modern telecommunications. All forms of travel would be more dangerous and expensive. Software engineering has contributed a great deal and I am convinced that its contributions in the 21st century will be even greater.



History of software engineering

The notion of 'software engineering' was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis' (Naur and Randell, 1969). It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding and object-oriented development. Tools and standard notations were developed and are now extensively used.

<http://www.SoftwareEngineering-9.com/Web/History/>

1.1 Professional software development

Lots of people write programs. People in business write spreadsheet programs to simplify their jobs, scientists and engineers write programs to process their experimental data, and hobbyists write programs for their own interest and enjoyment. However, the vast majority of software development is a professional activity where software is developed for specific business purposes, for inclusion in other devices, or as software products such as information systems, CAD systems, etc. Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals. It is maintained and changed throughout its life.

Software engineering is intended to support professional software development, rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development. To help you to get a broad view of what software engineering is about, I have summarized some frequently asked questions in Figure 1.1.

Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly. A professionally developed software system is often more than a single program. The system usually consists of a number of separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the structure of the system; user documentation, which explains how to use the system, and websites for users to download recent product information.

This is one of the important differences between professional and amateur software development. If you are writing a program for yourself, no one else will use it and you don't have to worry about writing program guides, documenting the program design, etc. However, if you are writing software that other people will use and other engineers will change then you usually have to provide additional information as well as the code of the program.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the Web made to software engineering?	The Web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Figure 1.1 Frequently asked questions about software

Software engineers are concerned with developing software products (i.e., software which can be sold to a customer). There are two kinds of software products:

1. *Generic products* These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to

buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages, and project-management tools. It also includes so-called vertical applications designed for some specific purpose such as library information systems, accounting systems, or systems for maintaining dental records.

2. *Customized (or bespoke) products* These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

An important difference between these types of software is that, in generic products, the organization that develops the software controls the software specification. For custom products, the specification is usually developed and controlled by the organization that is buying the software. The software developers must work to that specification.

However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as the SAP system, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

When we talk about the quality of professional software, we have to take into account that the software is used and changed by people apart from its developers. Quality is therefore not just concerned with what the software does. Rather, it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation. This is reflected in so-called quality or non-functional software attributes. Examples of these attributes are the software's response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on. These can be generalized into the set of attributes shown in Figure 1.2, which I believe are the essential characteristics of a professional software system.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. *Engineering discipline* Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Figure 1.2 Essential attributes of good software

and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.

2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

Engineering is about getting results of the required quality within the schedule and budget. This often involves making compromises—engineers cannot be perfectionists. People writing programs for themselves, however, can spend as much time as they wish on the program development.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances so a more creative, less formal approach to development may be effective in some circumstances. Less formal development is particularly appropriate for the development of web-based systems, which requires a blend of software and graphical design skills.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes. These activities are:

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Different types of systems need different development processes. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail depending on the type of software being developed. I describe software processes in more detail in Chapter 2.

Software engineering is related to both computer science and systems engineering:

1. Computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science cannot always be applied to large, complex problems that require a software solution.
2. System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.).

As I discuss in the next section, there are many different types of software. There is no universal software engineering method or technique that is applicable for all of these. However, there are three general issues that affect many different types of software:

1. *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.
2. *Business and social change* Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software. Many traditional software engineering techniques are time consuming and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.
3. *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot attack our software and that information security is maintained.

Of course, these are not independent issues. For example, it may be necessary to make rapid changes to a legacy system to provide it with a web service interface. To address these challenges we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

1.1.2 Software engineering diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. How this systematic approach is actually implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years.

Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application that is being developed. There are many different types of application including:

1. *Stand-alone applications* These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to

be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.

2. *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.
3. *Embedded control systems* These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
4. *Batch processing systems* These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.
5. *Entertainment systems* These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.
6. *Systems for modeling and simulation* These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.
7. *Data collection systems* These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.
8. *Systems of systems* These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

Of course, the boundaries between these system types are blurred. If you develop a game for a mobile (cell) phone, you have to take into account the same constraints (power, hardware interaction) as the developers of the phone software. Batch processing systems are often used in conjunction with web-based systems. For example,

in a company, travel expense claims may be submitted through a web application but processed in a batch application for monthly payment.

You use different software engineering techniques for each type of system because the software has quite different characteristics. For example, an embedded control system in an automobile is safety-critical and is burned into ROM when installed in the vehicle. It is therefore very expensive to change. Such a system needs very extensive verification and validation so that the chances of having to recall cars after sale to fix software problems are minimized. User interaction is minimal (or perhaps nonexistent) so there is no need to use a development process that relies on user interface prototyping.

For a web-based system, an approach based on iterative development and delivery may be appropriate, with the system being composed of reusable components. However, such an approach may be impractical for a system of systems, where detailed specifications of the system interactions have to be specified in advance so that each system can be separately developed.

Nevertheless, there are software engineering fundamentals that apply to all types of software system:

1. They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, different processes are used for different types of software.
2. Dependability and performance are important for all types of systems. Software should behave as expected, without failures and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
3. Understanding and managing the software specification and requirements (what the software should do) are important. You have to know what different customers and users of the system expect from it and you have to manage their expectations so that a useful system can be delivered within budget and to schedule.
4. You should make as effective use as possible of existing resources. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

These fundamental notions of process, dependability, requirements, management, and reuse are important themes of this book. Different methods reflect them in different ways but they underlie all professional software development.

You should notice that these fundamentals do not cover implementation and programming. I don't cover specific programming techniques in this book because these vary dramatically from one type of system to another. For example, a scripting language such as Ruby is used for web-based system programming but would be completely inappropriate for embedded systems engineering.

1.1.3 Software engineering and the Web

The development of the World Wide Web has had a profound effect on all of our lives. Initially, the Web was primarily a universally accessible information store and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organization. Around 2000, the Web started to evolve and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of a vast range of new system products that delivered innovative services, accessed over the Web. These are often funded by adverts that are displayed on the user's screen and do not involve direct payment from users.

As well as these system products, the development of web browsers that could run small programs and do some local processing led to an evolution in business and organizational software. Instead of writing software and deploying it on users' PCs, the software was deployed on a web server. This made it much cheaper to change and upgrade the software, as there was no need to install the software on every PC. It also reduced costs, as user interface development is particularly expensive. Consequently, wherever it has been possible to do so, many businesses have moved to web-based interaction with company software systems.

The next stage in the development of web-based systems was the notion of web services. Web services are software components that deliver specific, useful functionality and which are accessed over the Web. Applications are constructed by integrating these web services, which may be provided by different companies. In principle, this linking can be dynamic so that an application may use different web services each time that it is executed. I cover this approach to software development in Chapter 19.

In the last few years, the notion of 'software as a service' has been developed. It has been proposed that software will not normally run on local computers but will run on 'computing clouds' that are accessed over the Internet. If you use a service such as web-based mail, you are using a cloud-based system. A computing cloud is a huge number of linked computer systems that is shared by many users. Users do not buy software but pay according to how much the software is used or are given free access in return for watching adverts that are displayed on their screen.

The advent of the web, therefore, has led to a significant change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters. Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.

This radical change in software organization has, obviously, led to changes in the ways that web-based systems are engineered. For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.

2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems should be developed and delivered incrementally.
3. User interfaces are constrained by the capabilities of web browsers. Although technologies such as AJAX (Holdener, 2008) mean that rich interfaces can be created within a web browser, these technologies are still difficult to use. Web forms with local scripting are more commonly used. Application interfaces on web-based systems are often poorer than the specially designed user interfaces on PC system products.

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system. Experience gained with large system development in the 20th century is still relevant to web-based software.

1.2 Software engineering ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.

It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. *Confidentiality* You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.3 The ACM/IEEE Code of Ethics (© IEEE/ACM 1999)

Professional societies and institutions have an important role to play in setting ethical standards. Organizations such as the ACM, the IEEE (Institute of Electrical and Electronic Engineers), and the British Computer Society publish a code of professional conduct or code of ethics. Members of these organizations undertake to follow that code when they sign up for membership. These codes of conduct are generally concerned with fundamental ethical behavior.

Professional associations, notably the ACM and the IEEE, have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form, shown in Figure 1.3, and a longer form (Gotterbarn et al., 1999) that adds detail and substance to the shorter version. The rationale behind this code is summarized in the first two paragraphs of the longer form:

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software

systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer.

In any situation where different people have different views and objectives you are likely to be faced with ethical dilemmas. For example, if you disagree, in principle, with the policies of more senior management in the company, how should you react? Clearly, this depends on the particular individuals and the nature of the disagreement. Is it best to argue a case for your position from within the organization or to resign in principle? If you feel that there are problems with a software project, when do you reveal these to management? If you discuss these while they are just a suspicion, you may be overreacting to a situation; if you leave it too late, it may be impossible to resolve the difficulties.

Such ethical dilemmas face all of us in our professional lives and, fortunately, in most cases they are either relatively minor or can be resolved without too much difficulty. Where they cannot be resolved, the engineer is faced with, perhaps, another problem. The principled action may be to resign from their job but this may well affect others such as their partner or their children.

A particularly difficult situation for professional engineers arises when their employer acts in an unethical way. Say a company is responsible for developing a safety-critical system and, because of time pressure, falsifies the safety validation records. Is the engineer's responsibility to maintain confidentiality or to alert the customer or publicize, in some way, that the delivered system may be unsafe?

The problem here is that there are no absolutes when it comes to safety. Although the system may not have been validated according to predefined criteria, these criteria may be too strict. The system may actually operate safely throughout its lifetime. It is also the case that, even when properly validated, the system may fail and cause an accident. Early disclosure of problems may result in damage to the employer and other employees; failure to disclose problems may result in damage to others.

You must make up your own mind in these matters. The appropriate ethical position here depends entirely on the views of the individuals who are involved. In this case, the potential for damage, the extent of the damage, and the people affected by the damage should influence the decision. If the situation is very dangerous, it may be justified to publicize it using the national press (say). However, you should always try to resolve the situation while respecting the rights of your employer.

Another ethical issue is participation in the development of military and nuclear systems. Some people feel strongly about these issues and do not wish to participate in any systems development associated with military systems. Others will work on military systems but not on weapons systems. Yet others feel that national security is an overriding principle and have no ethical objections to working on weapons systems.

In this situation, it is important that both employers and employees should make their views known to each other in advance. Where an organization is involved in military or nuclear work, they should be able to specify that employees must be willing to accept any work assignment. Equally, if an employee is taken on and makes clear that they do not wish to work on such systems, employers should not put pressure on them to do so at some later date.

The general area of ethics and professional responsibility is becoming more important as software-intensive systems pervade every aspect of work and everyday life. It can be considered from a philosophical standpoint where the basic principles of ethics are considered and software engineering ethics are discussed with reference to these basic principles. This is the approach taken by Laudon (1995) and to a lesser extent by Huff and Martin (1995). Johnson's text on computer ethics (2001) also approaches the topic from a philosophical perspective.

However, I find that this philosophical approach is too abstract and difficult to relate to everyday experience. I prefer the more concrete approach embodied in codes of conduct and practice. I think that ethics are best discussed in a software engineering context and not as a subject in their own right. In this book, therefore, I do not include abstract ethical discussions but, where appropriate, include examples in the exercises that can be the starting point for a group discussion on ethical issues.

1.3 Case studies

To illustrate software engineering concepts, I use examples from three different types of systems throughout the book. The reason why I have not used a single case study is that one of the key messages in this book is that software engineering practice depends on the type of systems being produced. I therefore choose an appropriate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.

The three types of systems that I use as case studies are:

1. *An embedded system* This is a system where the software controls a hardware device and is embedded in that device. Issues in embedded systems typically

include physical size, responsiveness, power management, etc. The example of an embedded system that I use is a software system to control a medical device.

2. *An information system* This is a system whose primary purpose is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system that I use is a medical records system.
3. *A sensor-based data collection system* This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.

I introduce each of these systems in this chapter, with more information about each of them available on the Web.

1.3.1 An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

The problem with this treatment is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.

Current advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this already exist for the treatment of hospital patients. In the future, it may be possible for many diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system might work by using a micro-sensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

Figure 1.4 Insulin pump hardware

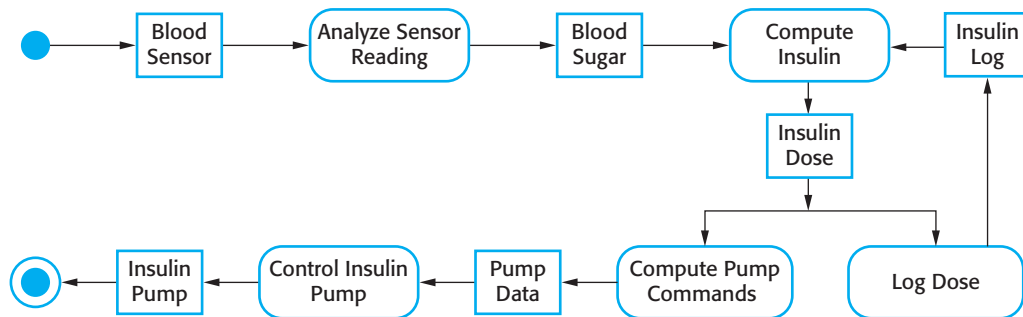
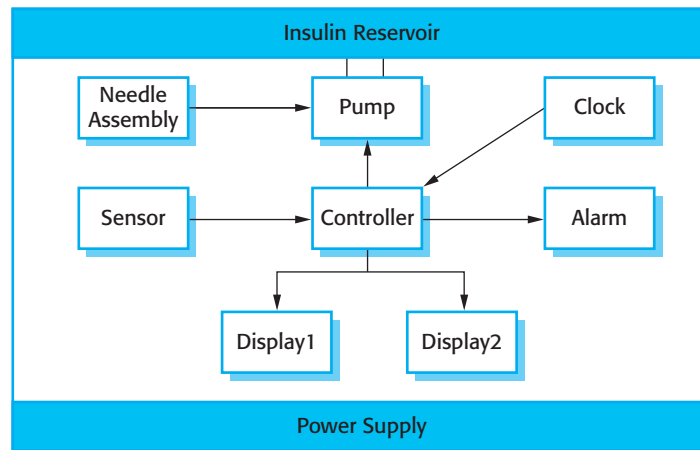


Figure 1.5 Activity model of the insulin pump

Figure 1.4 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a controller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Figure 1.5 is a UML activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. There are, therefore, two essential high-level requirements that this system must meet:

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

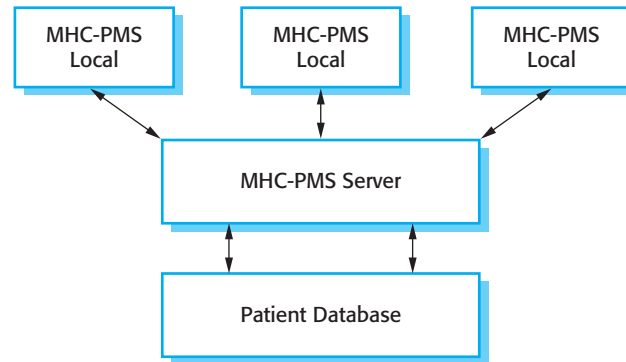


Figure 1.6 The organization of the MHC-PMS

The system must therefore be designed and implemented to ensure that the system always meets these requirements. More detailed requirements and discussions of how to ensure that the system is safe are discussed in later chapters.

1.3.2 A patient information system for mental health care

A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.

The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics. It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity. When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems. Figure 1.6 illustrates the organization of the MHC-PMS.

The MHC-PMS has two overall goals:

1. To generate management information that allows health service managers to assess performance against local and government targets.
2. To provide medical staff with timely information to support the treatment of patients.

The nature of mental health problems is such that patients are often disorganized so may miss appointments, deliberately or accidentally lose prescriptions and medication, forget instructions, and make unreasonable demands on medical staff. They may drop in on clinics unexpectedly. In a minority of cases, they may be a danger to themselves or to other people. They may regularly change address or may be homeless on a long-term or short-term basis. Where patients are dangerous, they may need to be ‘sectioned’—confined to a secure hospital for treatment and observation.

Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.

The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments. Reports are generated at regular intervals for medical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.

The key features of the system are:

1. *Individual care management* Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. *Patient monitoring* The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.
3. *Administrative reporting* The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system. These are laws on data protection that govern the confidentiality of personal information and mental health laws that govern the compulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to very strict legislative safeguards. One of the aims of the MHC-PMS is to ensure that staff always act in accordance with the law and that their decisions are recorded for judicial review if necessary.

As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves. The MHC-PMS is also a safety-critical

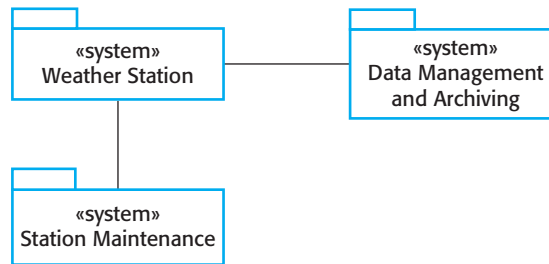


Figure 1.7 The weather station's environment

system. Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.

The overall design of the system has to take into account privacy and safety requirements. The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients. There is a potential conflict here—privacy is easiest to maintain when there is only a single copy of the system data. However, to ensure availability in the event of server failure or when disconnected from a network, multiple copies of the data should be maintained. I discuss the trade-offs between these requirements in later chapters.

1.3.3 A wilderness weather station

To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas. These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed, and wind direction.

Wilderness weather stations are part of a larger system (Figure 1.7), which is a weather information system that collects data from weather stations and makes it available to other systems for processing. The systems in Figure 1.7 are:

1. *The weather station system* This is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
2. *The data management and archiving system* This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.
3. *The station maintenance system* This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control a wilderness weather system.

In Figure 1.7, I have used the UML package symbol to indicate that each system is a collection of components and have identified the separate systems, using the UML stereotype «system». The associations between the packages indicate there is an exchange of information but, at this stage, there is no need to define them in any more detail.

Each weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure, and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

The weather station system operates by collecting weather observations at frequent intervals—for example, temperatures are measured every minute. However, because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data. It then transmits this aggregated data when requested by the data collection system. If, for whatever reason, it is impossible to make a connection, then the weather station maintains the data locally until communication can be resumed.

Each weather station is battery-powered and must be entirely self-contained—there are no external power or network cables available. All communications are through a relatively slow-speed satellite link and the weather station must include some mechanism (solar or wind power) to charge its batteries. As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals. The station software is therefore not just concerned with data collection. It must also:

1. Monitor the instruments, power, and communication hardware and report faults to the management system.
2. Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
3. Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

Because weather stations have to be self-contained and unattended, this means that the software installed is complex, even though the data collection functionality is fairly simple.

KEY POINTS

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software is not just a program or programs but also includes documentation. Essential software product attributes are maintainability, dependability, security, efficiency, and acceptability.
- The software process includes all of the activities involved in software development. The high-level activities of specification, development, validation, and evolution are part of all software processes.
- The fundamental notions of software engineering are universally applicable to all types of system development. These fundamentals include software processes, dependability, security, requirements, and reuse.
- There are many different types of systems and each requires appropriate software engineering tools and techniques for their development. There are few, if any, specific design and implementation techniques that are applicable to all kinds of systems.
- The fundamental ideas of software engineering are applicable to all types of software systems. These fundamentals include managed software processes, software dependability and security, requirements engineering, and software reuse.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.
- Professional societies publish codes of conduct that set out the standards of behavior expected of their members.

FURTHER READING

‘No silver bullet: Essence and accidents of software engineering’. In spite of its age, this paper is a good general introduction to the problems of software engineering. The essential message of the paper still hasn’t changed. (F. P. Brooks, *IEEE Computer*, **20** (4), April 1987.)
<http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532>.

‘Software engineering code of ethics is approved’. An article that discusses the background to the development of the ACM/IEEE Code of Ethics and that includes both the short and long form of the code. (*Comm. ACM*, D. Gotterbarn, K. Miller, and S. Rogerson, October 1999.)
<http://portal.acm.org/citation.cfm?doid=317665.317682>.

Professional Issues in Software Engineering. This is an excellent book discussing legal and professional issues as well as ethics. I prefer its practical approach to more theoretical texts on ethics. (F. Bott, A. Coleman, J. Eaton and D. Rowland, 3rd edition, 2000, Taylor and Francis.)

IEEE Software, March/April 2002. This is a special issue of the magazine devoted to the development of Web-based software. This area has changed very quickly so some articles are a little dated but most are still relevant. (*IEEE Software*, **19** (2), 2002.)
<http://www2.computer.org/portal/web/software>.

‘A View of 20th and 21st Century Software Engineering’. A backward and forward look at software engineering from one of the first and most distinguished software engineers. Barry Boehm identifies timeless software engineering principles but also suggests that some commonly used practices are obsolete. (B. Boehm, *Proc. 28th Software Engineering Conf.*, Shanghai. 2006.)
<http://doi.ieeecomputersociety.org/10.1145/1134285.1134288>.

‘Software Engineering Ethics’. Special issue of *IEEE Computer*, with a number of papers on the topic. (*IEEE Computer*, **42** (6), June 2009.)

EXERCISES

- 1.1. Explain why professional software is not just the programs that are developed for a customer.
- 1.2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?
- 1.3. What are the four important attributes that all professional software should have? Suggest four other attributes that may sometimes be significant.
- 1.4. Apart from the challenges of heterogeneity, business and social change, and trust and security, identify other problems and challenges that software engineering is likely to face in the 21st century (Hint: think about the environment).
- 1.5. Based on your own knowledge of some of the application types discussed in section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.
- 1.6. Explain why there are fundamental ideas of software engineering that apply to all types of software systems.
- 1.7. Explain how the universal use of the Web has changed software systems.
- 1.8. Discuss whether professional engineers should be certified in the same way as doctors or lawyers.
- 1.9. For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.3, suggest an appropriate example that illustrates that clause.
- 1.10. To help counter terrorism, many countries are planning or have developed computer systems that track large numbers of their citizens and their actions. Clearly this has privacy implications. Discuss the ethics of working on the development of this type of system.

REFERENCES

Gotterbarn, D., Miller, K. and Rogerson, S. (1999). Software Engineering Code of Ethics is Approved. *Comm. ACM*, **42** (10), 102–7.

Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Ca.: O'Reilly and Associates.

Huff, C. and Martin, C. D. (1995). Computing Consequences: A Framework for Teaching Ethical Computing. *Comm. ACM*, **38** (12), 75–84.

Johnson, D. G. (2001). *Computer Ethics*. Englewood Cliffs, NJ: Prentice Hall.

Laudon, K. (1995). Ethical Concepts and Information Technology. *Comm. ACM*, **38** (12), 33–9.

Naur, P. and Randell, B. (1969). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany. 7th to 11th October 1968.



2

Software processes

Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter you will:

- understand the concepts of software processes and software process models;
- have been introduced to three generic software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

Contents

- 2.1** Software process models
- 2.2** Process activities
- 2.3** Coping with change
- 2.4** The Rational Unified Process

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

In some form, these activities are part of all software processes. In practice, of course, they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There are also supporting process activities such as documentation and software configuration management.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

1. Products, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no ideal process and most organizations have developed their own software development processes. Processes have evolved to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed. For some

systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

Sometimes, software processes are categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and it is easier to change the process to reflect changing customer requirements. As Boehm and Turner (2003) discuss, each approach is suitable for different types of software. Generally, you need to find a balance between plan-driven and agile processes.

Although there is no ‘ideal’ software process, there is scope for improving the software process in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development.

Software processes can be improved by process standardization where the diversity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time, and makes automated process support more economical. Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice. I discuss software process improvement in more detail in Chapter 26.

2.1 Software process models

As I explained in Chapter 1, a software process model is a simplified representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called ‘process paradigms’) and present these from an architectural perspective. That is, we see the framework of the process but not the details of specific activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.

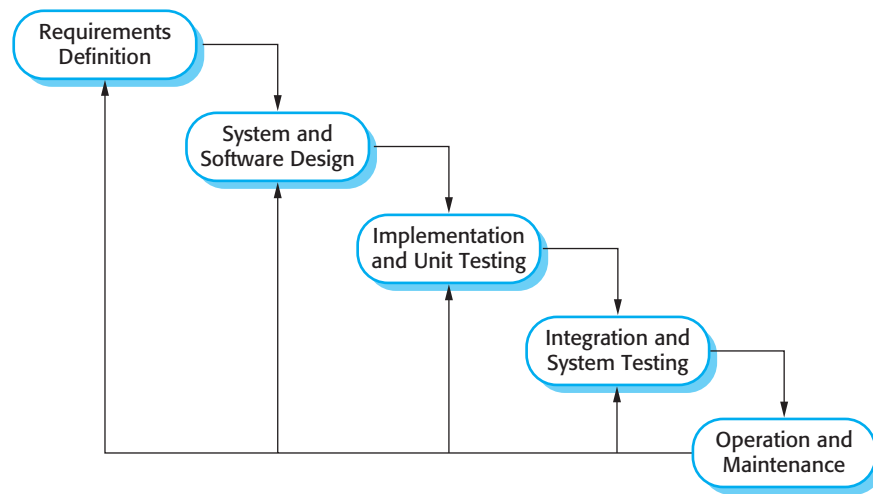


Figure 2.1 The waterfall model

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. *Reuse-oriented software engineering* This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

These models are not mutually exclusive and are often used together, especially for large systems development. For large systems, it makes sense to combine some of the best features of the waterfall and the incremental development models. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Sub-systems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process. Parts of the system which are difficult to specify in advance, such as the user interface, should always be developed using an incremental approach.

2.1.1 The waterfall model

The first published model of the software development process was derived from more general system engineering processes (Royce, 1970). This model is illustrated in Figure 2.1. Because of the cascade from one phase to another, this model is known as the ‘waterfall model’ or software life cycle. The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them.

The principal stages of the waterfall model directly reflect the fundamental development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved ('signed off'). The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified. During coding, design problems are found and so on. The software process is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.



Cleanroom software engineering

An example of a formal development process, originally developed by IBM, is the Cleanroom process. In the Cleanroom process each software increment is formally specified and this specification is transformed into an implementation. Software correctness is demonstrated using a formal approach. There is no unit testing for defects in the process and the system testing is focused on assessing the system's reliability.

The objective of the Cleanroom process is zero-defects software so that delivered systems have a high level of reliability.

<http://www.SoftwareEngineering-9.com/Web/Cleanroom/>

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification.

Formal development processes, such as that based on the B method (Schneider, 2001; Wordsworth, 1996) are particularly suited to the development of systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements.

Processes based on formal transformations are generally only used in the development of safety-critical or security-critical systems. They require specialized expertise. For the majority of systems this process does not offer significant cost-benefits over other approaches to system development.

2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 2.2). Specification, development, and

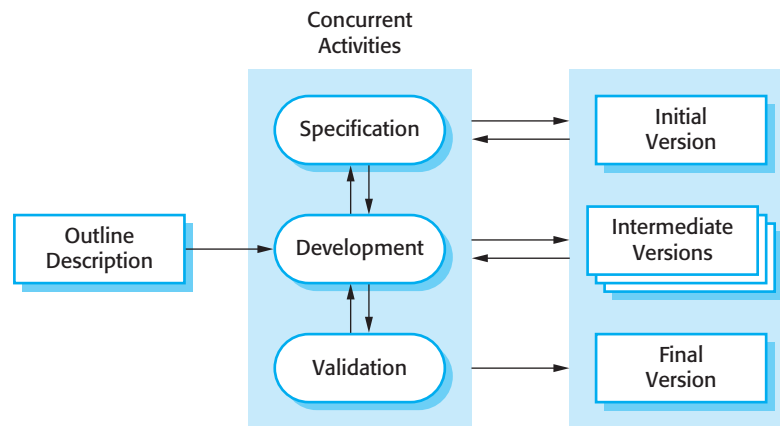


Figure 2.2 Incremental development

validation activities are interleaved rather than separate, with rapid feedback across activities.

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.
3. More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

**Problems with incremental development**

Although incremental development has many advantages, it is not problem-free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons—for example, there may be procedures to ensure that the software properly implements external regulations (e.g., in the United States, the Sarbanes-Oxley accounting regulations). Changing these procedures may not be possible so process conflicts may be unavoidable.

<http://www.SoftwareEngineering-9.com/Web/IncrementalDev/>

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

You can develop a system incrementally and expose it to customers for comment, without actually delivering it and deploying it in the customer's environment. Incremental delivery and deployment means that the software is used in real, operational processes. This is not always possible as experimenting with new software can disrupt normal business processes. I discuss the advantages and disadvantages of incremental delivery in Section 2.3.2.

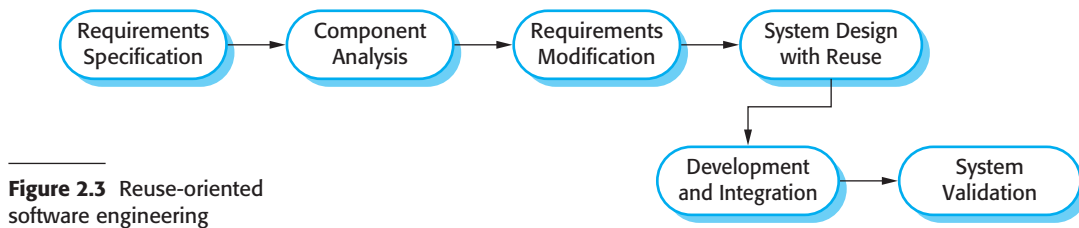


Figure 2.3 Reuse-oriented software engineering

2.1.3 Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure 2.3. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse-oriented process are different. These stages are:

1. *Component analysis* Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
2. *Requirements modification* During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
3. *System design with reuse* During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
4. *Development and integration* Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.

Reuse-oriented software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important and I have dedicated several chapters in the third part of the book to this topic. General issues of software reuse and COTS reuse are covered in Chapter 16, component-based software engineering in Chapters 17 and 18, and service-oriented systems in Chapter 19.

2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Software developers use a variety of different software tools in their work. Tools are particularly useful for supporting the editing of different types of document and for managing the immense volume of detailed information that is generated in a large software project.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved. In extreme programming, for example, specifications are written on cards. Tests are executable and developed before the program itself. Evolution may involve substantial system restructuring or refactoring.

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a



Software development tools

Software development tools (sometimes called Computer-Aided Software Engineering or CASE tools) are programs that are used to support software engineering process activities. These tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. Examples of activities that can be automated include:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way. The ECLIPSE IDE is widely used and has been designed to incorporate many different types of software tools.

<http://www.SoftwareEngineering-9.com/Web/CASE/>

particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

There are four main activities in the requirements engineering process:

1. *Feasibility study* An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.
2. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
3. *Requirements specification* Requirements specification is the activity of translating the information gathered during the analysis activity into a document that

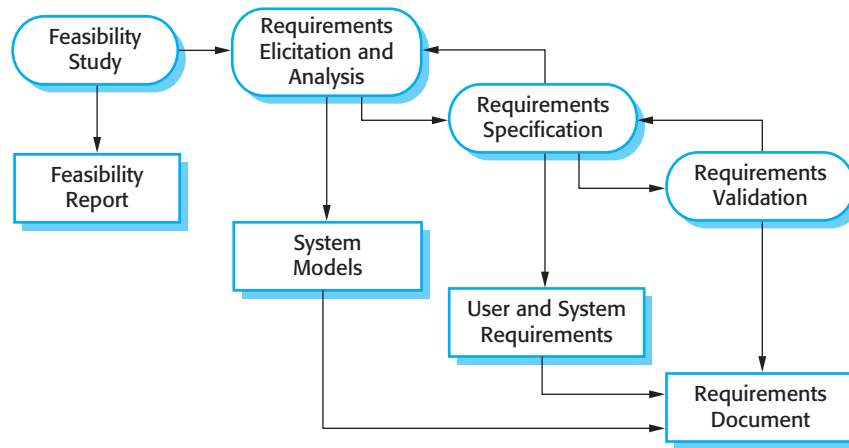


Figure 2.4 The requirements engineering process

defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

4. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Of course, the activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved. In agile methods, such as extreme programming, requirements are developed incrementally according to user priorities and the elicitation of requirements comes from users who are part of the development team.

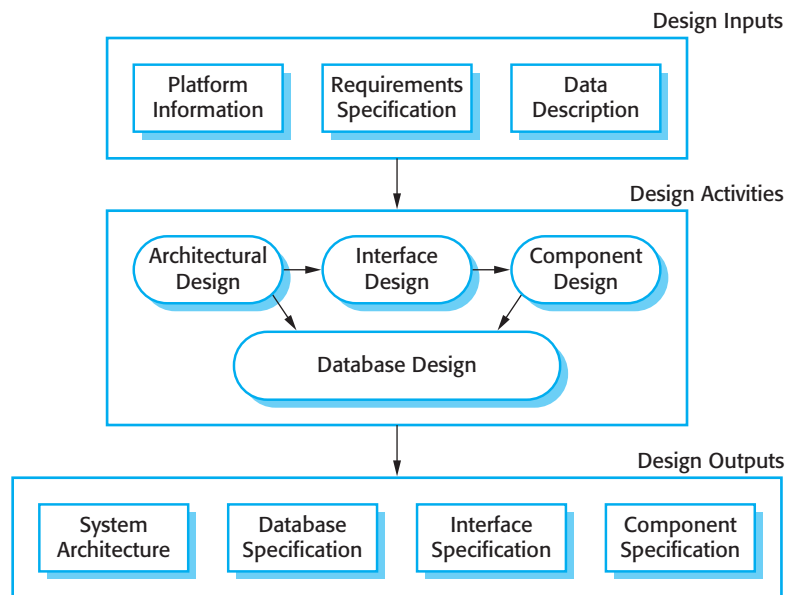
2.2.2 Software design and implementation

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

Figure 2.5 is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

Figure 2.5 A general model of the design process



The diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

Most software interfaces with other software systems. These include the operating system, database, middleware, and other application systems. These make up the ‘software platform’, the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with the software’s environment. The requirements specification is a description of the functionality the software must provide and its performance and dependability requirements. If the system is to process existing data, then the description of that data may be included in the platform specification; otherwise, the data description must be an input to the design process so that the system data organization to be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. *Interface design*, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.



Structured methods

Structured methods are an approach to software design in which graphical models that should be developed as part of the design process are defined. The method may also define a process for developing the models and rules that apply to each model type. Structured methods lead to standardized documentation for a system and are particularly useful in providing a development framework for less-experienced and less-expert software developers.

<http://www.SoftwareEngineering-9.com/Web/Structured-methods/>

3. *Component design*, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. *Database design*, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

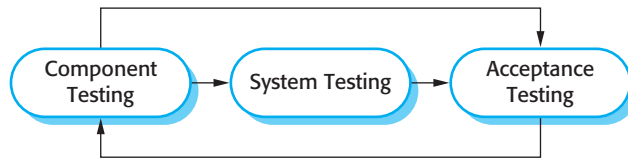
These activities lead to a set of design outputs, which are also shown in Figure 2.5. The detail and representation of these vary considerably. For critical systems, detailed design documents setting out precise and accurate descriptions of the system must be produced. If a model-driven approach is used, these outputs may mostly be diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

Structured methods for design were developed in the 1970s and 1980s and were the precursor to the UML and object-oriented design (Budgen, 2003). They rely on producing graphical models of the system and, in many cases, automatically generating code from these models. Model-driven development (MDD) or model-driven engineering (Schmidt, 2006), where models of the software are created at different levels of abstraction, is an evolution of structured methods. In MDD, there is greater emphasis on architectural models with a separation between abstract implementation-independent models and implementation-specific models. The models are developed in sufficient detail so that the executable system can be generated from them. I discuss this approach to development in Chapter 5.

The development of a program to implement the system follows naturally from the system design processes. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for the later stages of design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is a personal activity and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less-understood components. Others take the opposite

Figure 2.6 Stages of testing



approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects that must be removed from the program. This is called debugging. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, may be used to support the debugging process.

2.2.3 Software validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

The stages in the testing process are:

1. *Development testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions

or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.

2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form sub-systems that are individually tested before these sub-systems are themselves integrated to form the final system.
3. *Acceptance testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.

Normally, component development and testing processes are interleaved. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach, as the programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In extreme programming, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V).

Acceptance testing is sometimes called 'alpha testing'. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.

When a system is to be marketed as a software product, a testing process called 'beta testing' is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale.

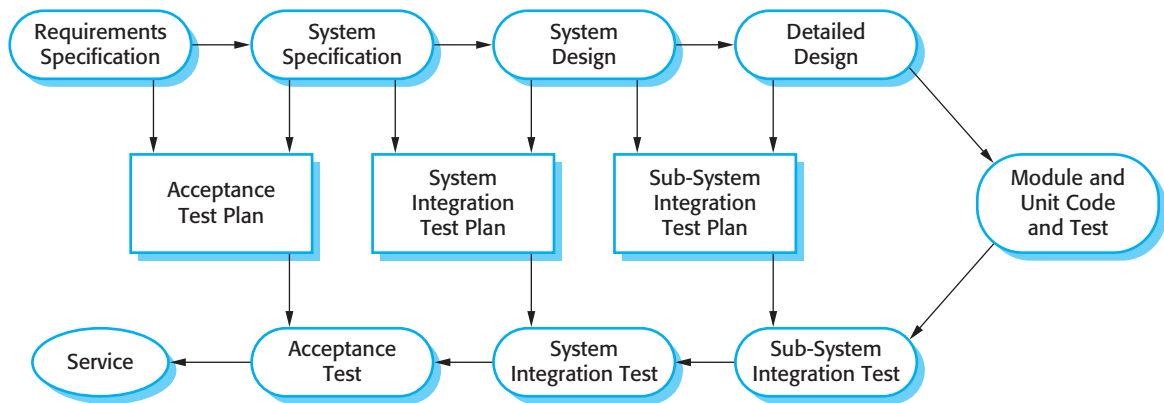


Figure 2.7 2.2.4 Software evolution
Testing phases
in a plan-driven
software process

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures and management priorities change. As new technologies become available, new design and implementation possibilities emerge. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

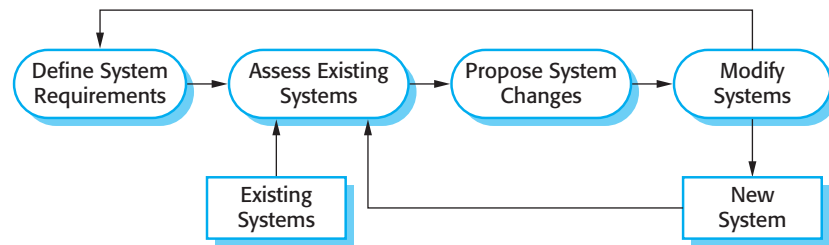


Figure 2.8 System evolution

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and re-test the system.

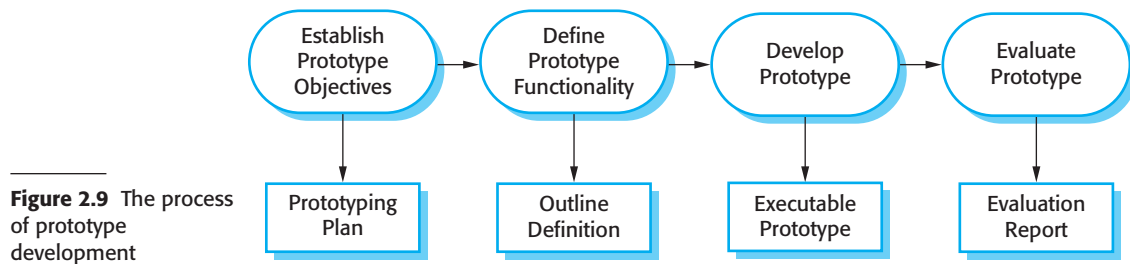
There are two related approaches that may be used to reduce the costs of rework:

1. Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements. These are:

1. System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.
2. Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3, which covers agile methods.



2.3.1 Prototyping

A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.

System prototypes allow users to see how well the system supports their work. They may get new ideas for requirements, and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the requirements that have been proposed. A function described in a specification may seem useful and well defined. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements.

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Prototyping is also an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, rapid prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the feasibility

of the application to managers. The same prototype cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions.

A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system. The tester of the prototype may not be typical of system users. The training time during prototype evaluation may be insufficient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

Developers are sometimes pressured by managers to deliver throwaway prototypes, particularly when there are delays in delivering the final version of the software. However, this is usually unwise:

1. It may be impossible to tune the prototype to meet non-functional requirements, such as performance, security, robustness, and reliability requirements, which were ignored during prototype development.
2. Rapid change during development inevitably means that the prototype is undocumented. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organizational quality standards are normally relaxed for prototype development.

Prototypes do not have to be executable to be useful. Paper-based mock-ups of the system user interface (Rettig, 1994) can be effective in helping users refine an interface design and work through usage scenarios. These are very cheap to develop and can be constructed in a few days. An extension of this technique is a Wizard of Oz prototype where only the user interface is developed. Users interact with this interface but their requests are passed to a person who interprets them and outputs the appropriate response.

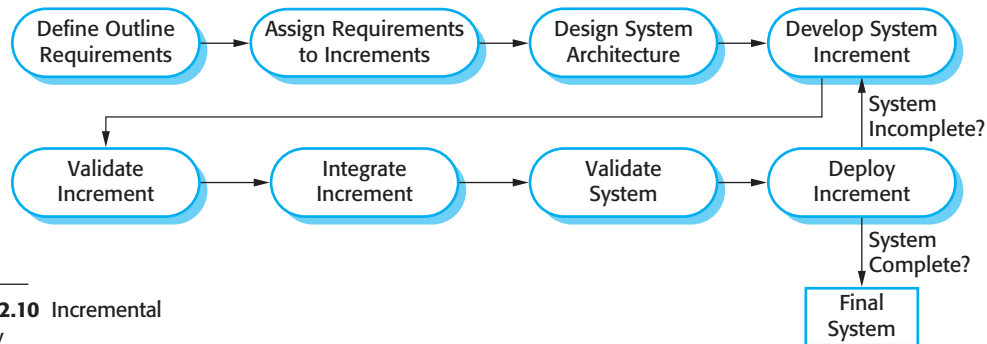


Figure 2.10 Incremental delivery

2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment. In an incremental delivery process, customers identify, in outline, the services to be provided by the system. They identify which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality. The allocation of services to increments depends on the service priority, with the highest-priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, customers can put it into service. This means that they take early delivery of part of the system functionality. They can experiment with the system and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system so there is no re-learning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.
4. As the highest-priority services are delivered first and increments then integrated, the most important system services receive the most testing. This means

that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery:

1. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
2. Iterative development can also be difficult when a replacement system is being developed. Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting useful customer feedback is difficult.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

There are some types of system where incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a process may be used in which a system prototype is developed iteratively and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

2.3.3 Boehm's spiral model

A risk-driven software process framework (the spiral model) was proposed by Boehm (1988). This is shown in Figure 2.11. Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that

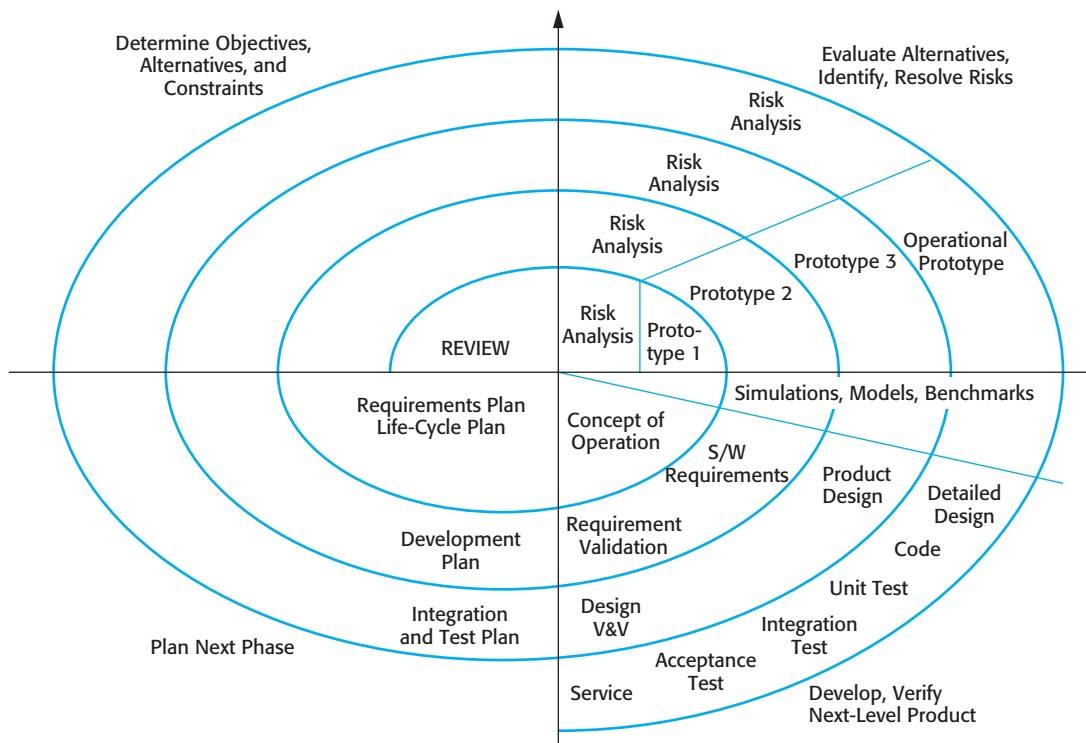


Figure 2.11 Boehm's spiral model of the software process (©IEEE 1988)

changes are a result of project risks and includes explicit risk management activities to reduce these risks.

Each loop in the spiral is split into four sectors:

1. *Objective setting* Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. *Risk assessment and reduction* For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. *Development and validation* After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use.
4. *Planning* The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives, and dealing with the constraints on each of them, are then enumerated. Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation.

Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity. Risk management, an essential part of project management, is covered in Chapter 22.

2.4 The Rational Unified Process

The Rational Unified Process (RUP) (Krutchen, 2003) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process (Rumbaugh, et al., 1999; Arlow and Neustadt, 2005). I have included a description here, as it is a good example of a hybrid process model. It brings together elements from all of the generic process models (Section 2.1), illustrates good practice in specification and design (Section 2.2) and supports prototyping and incremental delivery (Section 2.3).

The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Most descriptions of the RUP attempt to combine the static and dynamic perspectives in a single diagram (Krutchen, 2003). I think that makes the process harder to understand, so I use separate descriptions of each of these perspectives.

The RUP is a phased model that identifies four discrete phases in the software process. However, unlike the waterfall model where phases are equated with process activities, the phases in the RUP are more closely related to business rather than technical concerns. Figure 2.11 shows the phases in the RUP. These are:

1. *Inception* The goal of the inception phase is to establish a business case for the system. You should identify all external entities (people and systems) that will

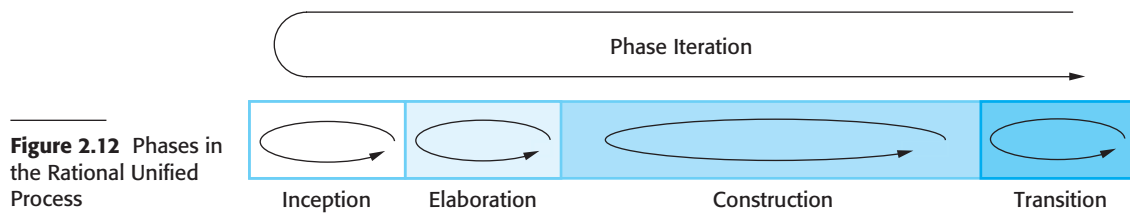


Figure 2.12 Phases in the Rational Unified Process

interact with the system and define these interactions. You then use this information to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.

2. *Elaboration* The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. On completion of this phase you should have a requirements model for the system, which may be a set of UML use-cases, an architectural description, and a development plan for the software.
3. *Construction* The construction phase involves system design, programming, and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.
4. *Transition* The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity. On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception in Figure 2.12.

The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting workflows. The RUP has been designed in conjunction with the UML, so the workflow description is oriented around associated UML models such as sequence models, object models, etc. The core engineering and support workflows are described in Figure 2.13.

The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. In the early phases of the process, most effort will probably be spent on workflows such as business modelling and requirements and, in the later phases, in testing and deployment.

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure 2.13 Static workflows in the Rational Unified Process

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. *Develop software iteratively* Plan increments of the system based on customer priorities and develop the highest-priority system features early in the development process.
2. *Manage requirements* Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. *Use component-based architectures* Structure the system architecture into components, as discussed earlier in this chapter.
4. *Visually model software* Use graphical UML models to present static and dynamic views of the software.
5. *Verify software quality* Ensure that the software meets the organizational quality standards.

6. *Control changes to software* Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP is not a suitable process for all types of development, e.g., embedded software development. However, it does represent an approach that potentially combines the three generic process models discussed in Section 2.1. The most important innovations in the RUP are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reuse-oriented development.
- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system. Systematic design methods may be used as part of this transformation.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction, and transition) but separates activities (requirements, analysis, and design, etc.) from these phases.

FURTHER READING

Managing Software Quality and Business Risk. This is primarily a book about software management but it includes an excellent chapter (Chapter 4) on process models. (M. Ould, John Wiley and Sons Ltd, 1999.)

Process Models in Software Engineering. This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering*, ed. J.J. Marciniak, John Wiley and Sons, 2001.) <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>.

The Rational Unified Process—An Introduction (3rd edition). This is the most readable book available on the RUP at the time of this writing. Krutchen describes the process well, but I would like to have seen more on the practical difficulties of using the process. (P. Krutchen, Addison-Wesley, 2003.)

EXERCISES

- 2.1. Giving reasons for your answer based on the type of system being developed, suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems:
 - A system to control anti-lock braking in a car
 - A virtual reality system to support software maintenance
 - A university accounting system that replaces an existing system
 - An interactive travel planning system that helps users plan journeys with the lowest environmental impact
- 2.2. Explain why incremental development is the most effective approach for developing business software systems. Why is this model less appropriate for real-time systems engineering?
- 2.3. Consider the reuse-based process model shown in Figure 2.3. Explain why it is essential to have two separate requirements engineering activities in the process.
- 2.4. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.
- 2.5. Describe the main activities in the software design process and the outputs of these activities. Using a diagram, show possible relationships between the outputs of these activities.
- 2.6. Explain why change is inevitable in complex systems and give examples (apart from prototyping and incremental delivery) of software process activities that help predict changes and make the software being developed more resilient to change.

- 2.7. Explain why systems developed as prototypes should not normally be used as production systems.
- 2.8. Explain why Boehm's spiral model is an adaptable model that can support both change avoidance and change tolerance activities. In practice, this model has not been widely used. Suggest why this might be the case.
- 2.9. What are the advantages of providing static and dynamic views of the software process as in the Rational Unified Process?
- 2.10. Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

REFERENCES

- Arlow, J. and Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Boston: Addison-Wesley.
- Boehm, B. and Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.
- Boehm, B. W. (1988). 'A Spiral Model of Software Development and Enhancement'. *IEEE Computer*, **21** (5), 61–72.
- Budgen, D. (2003). *Software Design (2nd Edition)*. Harlow, UK.: Addison-Wesley.
- Krutchén, P. (2003). *The Rational Unified Process—An Introduction (3rd Edition)*. Reading, MA: Addison-Wesley.
- Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.
- Rettig, M. (1994). 'Practical Programmer: Prototyping for Tiny Fingers'. *Comm. ACM*, **37** (4), 21–7.
- Royce, W. W. (1970). 'Managing the Development of Large Software Systems: Concepts and Techniques'. IEEE WESTCON, Los Angeles CA: 1–9.
- Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley.
- Schmidt, D. C. (2006). 'Model-Driven Engineering'. *IEEE Computer*, **39** (2), 25–31.
- Schneider, S. (2001). *The B Method*. Houndmills, UK: Palgrave Macmillan.
- Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.



3

Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know the key practices in extreme programming and how these relate to the general principles of agile methods;
- understand the Scrum approach to agile project management;
- be aware of the issues and problems of scaling agile development methods to the development of large software systems.

Contents

- 3.1** Agile methods
- 3.2** Plan-driven and agile development
- 3.3** Extreme programming
- 3.4** Agile project management
- 3.5** Scaling agile methods

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions, and the emergence of competing products and services. Software is part of almost all business operations so new software is developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid development and delivery is therefore now often the most critical requirement for software systems. In fact, many businesses are willing to trade off software quality and compromise on requirements to achieve faster deployment of the software that they need.

Because these businesses are operating in a changing environment, it is often practically impossible to derive a complete set of stable software requirements. The initial requirements inevitably change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, the requirements are likely to change quickly and unpredictably due to external factors. The software may then be out of date when it is delivered.

Software development processes that plan on completely specifying the requirements and then designing, building, and testing the system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually prolonged and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, a plan-driven approach is the right one. However, in a fast-moving business environment, this can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid system development and processes that can handle changing requirements has been recognized for some time. IBM introduced incremental development in the 1980s (Mills et al., 1980). The introduction of so-called fourth-generation languages, also in the 1980s, supported the idea of quickly developing and delivering software (Martin, 1981). However, the notion really took off in the late 1990s with the development of the notion of agile approaches such as DSDM (Stapleton, 1997), Scrum (Schwaber and Beedle, 2001), and extreme programming (Beck, 1999; Beck, 2000).

Rapid software development processes are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design, and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to

implement the system. The user requirements document only defines the most important characteristics of the system.

2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version. They may propose changes to the software and new requirements that should be implemented in a later version of the system.
3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface. The system may then generate a web-based interface for a browser or an interface for a specific platform such as Microsoft Windows.

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, the use of analysis and design methods supported by CASE tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This software was developed by large teams working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. These plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design has to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led a number of software developers in the 1990s to propose new 'agile methods'. These allowed the development team to focus on the software itself rather than on its design

and documentation. Agile methods universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto that was agreed on by many of the leading developers of these methods. This manifesto states:

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.

Probably the best-known agile method is extreme programming (Beck, 1999; Beck, 2000), which I describe later in this chapter. Other agile approaches include Scrum (Cohn, 2009; Schwaber, 2004; Schwaber and Beedle, 2001), Crystal (Cockburn, 2001; Cockburn, 2004), Adaptive Software Development (Highsmith, 2000), DSDM (Stapleton, 1997; Stapleton, 2003), and Feature Driven Development (Palmer and Felsing, 2002). The success of these methods has led to some integration with more traditional development methods based on system modelling, resulting in the notion of agile modelling (Ambler and Jeffries, 2002) and agile instantiations of the Rational Unified Process (Larman, 2002).

Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this. However, they share a set of principles, based on the agile manifesto, and so have much in common. These principles are shown in Figure 3.1. Different agile methods instantiate these principles in different ways and I don't have space to discuss all agile methods. Instead, I focus on two of the most widely used methods: extreme programming (Section 3.3) and Scrum (Section 3.4).

Agile methods have been very successful for some types of system development:

1. Product development where a software company is developing a small or medium-sized product for sale.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 3.1 The principles of agile methods

As I discuss in the final section of this chapter, the success of agile methods has meant that there is a lot of interest in using these methods for other types of software development. However, because of their focus on small, tightly integrated teams, there are problems in scaling them to large systems. There have also been experiments in using agile approaches for critical systems engineering (Drobna et al., 2004). However, because of the need for security, safety, and dependability analysis in critical systems, agile methods require significant modification before they can be routinely used for critical systems engineering.

In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.
2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
3. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.

5. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

Another non-technical problem—that is a general problem with incremental development and delivery—occurs when the system customer uses an outside organization for system development. The software requirements document is usually part of the contract between the customer and the supplier. Because incremental specification is inherent in agile methods, writing contracts for this type of development may be difficult.

Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. So long as all goes well, this benefits both the customer and the developer. However, if problems arise then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

Most books and papers that describe agile methods and experiences with agile methods talk about the use of these methods for new systems development. However, as I explain in Chapter 9, a huge amount of software engineering effort goes into the maintenance and evolution of existing software systems. There are only a small number of experience reports on using agile methods for software maintenance (Poole and Huisman, 2001). There are two questions that should be considered when considering agile methods and maintenance:

1. Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
2. Can agile methods be used effectively for evolving a system in response to customer change requests?

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is often not kept up to date and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. Agile practices therefore emphasize the importance of writing well-structured code and investing effort in code improvement. Therefore, the lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance suggests that the key document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. In this

respect, the use of agile methods is likely to make subsequent system maintenance more difficult and expensive.

Agile practices, used in the maintenance process itself, are likely to be effective, whether or not an agile approach has been used for system development. Incremental delivery, design for change and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process of software evolution.

However, the main difficulty after software delivery is likely to be keeping customers involved in the process. Although a customer may be able to justify the full-time involvement of a representative during system development, this is less likely during maintenance where changes are not continuous. Customer representatives are likely to lose interest in the system. Therefore, it is likely that alternative mechanisms, such as change proposals, discussed in Chapter 25, will be required to create the new system requirements.

The other problem that is likely to arise is maintaining continuity of the development team. Agile methods rely on team members understanding aspects of the system without having to consult documentation. If an agile development team is broken up, then this implicit knowledge is lost and it is difficult for new team members to build up the same understanding of the system and its components.

Supporters of agile methods have been evangelical in promoting their use and have tended to overlook their shortcomings. This has prompted an equally extreme response, which, in my view, exaggerates the problems with this approach (Stephens and Rosenberg, 2003). More reasoned critics such as DeMarco and Boehm (DeMarco and Boehm, 2002) highlight both the advantages and disadvantages of agile methods. They propose a hybrid approach where agile methods incorporate some techniques from plan-driven development may be the best way forward.

3.2 Plan-driven and agile development

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity. Figure 3.2 shows the distinctions between plan-driven and agile approaches to system specification.

In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process. For example, the requirements will evolve and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together, rather than separately.

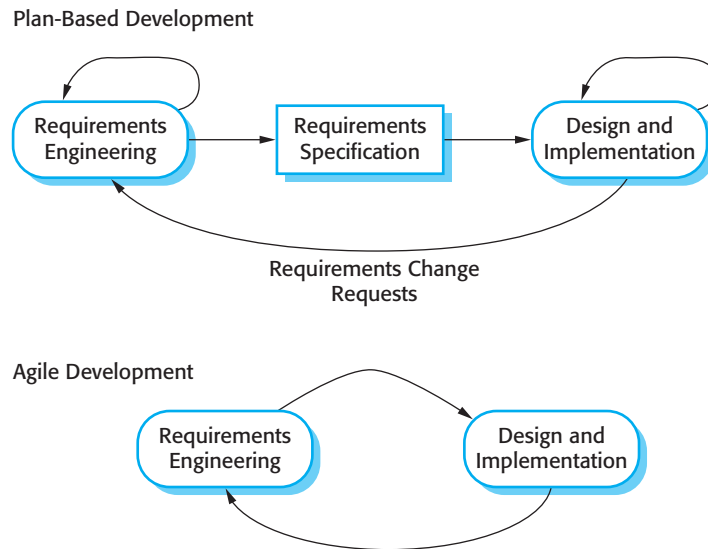


Figure 3.2 Plan-driven and agile specification

A plan-driven software process can support incremental development and delivery. It is perfectly feasible to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused and it may produce some design documentation. As I discuss in the following section, the agile development team may decide to include a documentation ‘spike’, where, instead of producing a new version of a system, the team produce system documentation.

In fact, most software projects include practices from plan-driven and agile approaches. To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human, and organizational questions:

1. Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
2. Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
3. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.
4. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
5. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system

developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and it is not of much use for long-term system maintenance.

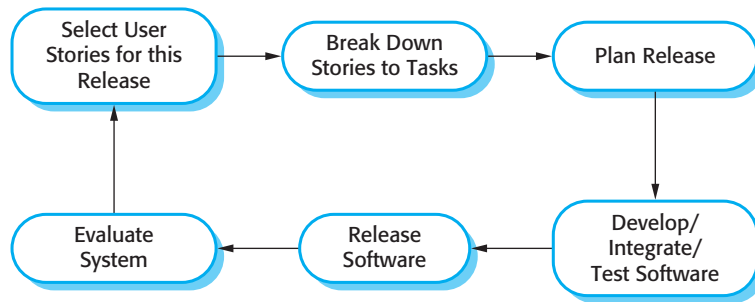
6. What technologies are available to support system development? Agile methods often rely on good tools to keep track of an evolving design. If you are developing a system using an IDE that does not have good tools for program visualization and analysis, then more design documentation may be required.
7. How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams. You may need to plan in advance what these are.
8. Are there cultural issues that may affect the system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering. This usually requires extensive design documentation, rather than the informal knowledge used in agile processes.
9. How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code. If you have a team with relatively low skill levels, you may need to use the best people to develop the design, with others responsible for programming.
10. Is the system subject to external regulation? If a system has to be approved by an external regulator (e.g., the Federal Aviation Authority [FAA] approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.

In reality, the issue of whether a project can be labelled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. In practice, many companies who claim to have used agile methods have adopted some agile practices and have integrated these with their plan-driven processes.

3.3 Extreme programming

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (2000) because the approach was developed by pushing recognized good practice, such as iterative development, to 'extreme' levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated and tested in a day.

Figure 3.3 The extreme programming release cycle



In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

Extreme programming involves a number of practices, summarized in Figure 3.4, which reflect the principles of agile methods:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system functions. Rather, the system customer is part of the development team and discusses scenarios with other team members. Together, they develop a ‘story card’ that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the mental

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Figure 3.4 Extreme programming practices

health care patient management system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

The story cards are the main inputs to the XP planning process or the 'planning game'. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

Prescribing Medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select current medication, 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose. If she wants to change the dose, she enters the dose and then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose and then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose and then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A
'prescribing medication'
story.

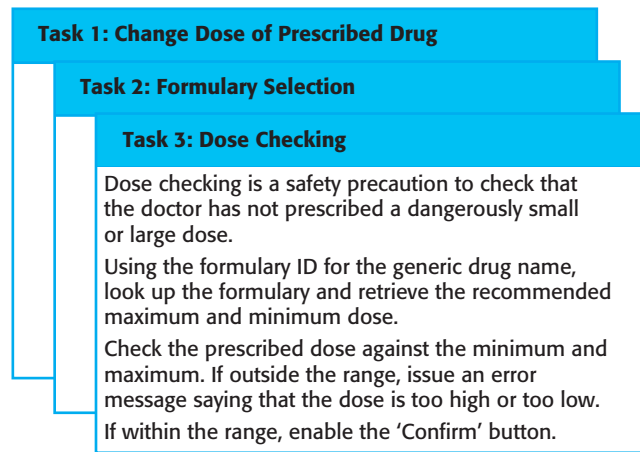
Sometimes, during the planning game, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. In XP terms, this is a 'spike', an increment where no programming is done. There may also be 'spikes' to design the system architecture or to develop system documentation.

Extreme programming takes an 'extreme' approach to incremental development. New versions of the software may be built several times per day and releases are delivered to customers roughly every two weeks. Release deadlines are never slipped; if there are development problems, the customer is consulted and functionality is removed from the planned release.

When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully. This then becomes the basis for the next iteration of the system.

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. The changes anticipated often never materialize and completely different change requests may actually be made. Therefore, the XP approach accepts that changes will happen and reorganize the software when these changes actually occur.

Figure 3.6 Examples of task cards for prescribing medication.



A general problem with incremental development is that it tends to degrade the software structure, so changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system.

Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team look for possible improvements to the software and implement them immediately. When a team member sees code that can be improved, they make these improvements even in situations where there is no immediate need for them. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of code with calls to methods defined in a program library. Program development environments, such as Eclipse (Carlson, 2005), include tools for refactoring which simplify the process of finding dependencies between code sections and making global code modifications.

In principle then, the software should always be easy to understand and change as new stories are implemented. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require the architecture of the system to be modified.

In practice, many companies that have adopted XP do not use all of the extreme programming practices listed in Figure 3.4. They pick and choose according to their local ways of working. For example, some companies find pair programming helpful; others prefer to use individual programming and reviews. To accommodate different levels of skill, some programmers don't do refactoring in parts of the system they did not develop, and conventional requirements may be used rather than user stories. However, most companies who have adopted an XP variant use small releases, test-first development, and continuous integration.

3.3.1 Testing in XP

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system.

The key features of testing in XP are:

1. Test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of ‘test-lag’. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 3.5) are shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Figure 3.7 Test case description for dose checking

Test 4: Dose Checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose \times frequency is too high and too low.
4. Test for inputs where single dose \times frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I discuss in Chapter 8, acceptance testing is the process where the system is tested using customer data to check that it meets the customer's real needs.

In XP, acceptance testing, like development, is incremental. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. For the story in Figure 3.5, the acceptance test would involve scenarios where (a) the dose of a drug was changed, (b) a new drug was selected, and (c) the formulary was used to find a drug. In practice, a series of acceptance tests rather than a single test are normally required.

Relying on the customer to support acceptance test development is sometimes a major difficulty in the XP testing process. People adopting the customer role have very limited available time and may not be able to work full-time with the development team. The customer may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be standalone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Massol and Husted, 2003) is a widely used example of an automated testing framework.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the ‘display logic’ and workflow between screens.
3. It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests written after development, then undetected bugs may be delivered in the system release.

3.3.2 Pair programming

Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. However, the same pairs do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

The use of pair programming has a number of advantages:

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg’s (1971) idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (covered in Chapter 24) are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Although pair programming is a less formal process that probably doesn’t find as many errors as code inspections, it is a much cheaper inspection process than formal program inspections.
3. It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as

two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers, Williams and her collaborators (Cockburn and Williams, 2001; Williams et al., 2000) found that productivity with pair programming seems to be comparable with that of two people working independently. The reasons suggested are that pairs discuss the software before development so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers (Arisholm et al., 2007; Parrish et al., 2004) did not replicate these results. They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.4 Agile project management

The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project. They supervise the work of software engineers and monitor how well the software development is progressing.

The standard approach to project management is plan-driven. As I discuss in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach really requires a manager to have a stable view of everything that has to be developed and the development processes. However, it does not work well with agile methods where the requirements are developed incrementally; where the software is delivered in short, rapid increments; and where changes to the requirements and the software are the norm.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to the team. This requires a different approach to project management, which is adapted to incremental development and the particular strengths of agile methods.

The Scrum approach (Schwaber, 2004; Schwaber and Beedle, 2001) is a general agile method but its focus is on managing iterative development rather than specific technical approaches to agile software engineering. Figure 3.8 is a diagram of the Scrum management process. Scrum does not prescribe the use of programming practices such as pair programming and test-first development. It can therefore be used with more technical agile approaches, such as XP, to provide a management framework for the project.

There are three phases in Scrum. The first is an outline planning phase where you establish the general objectives for the project and design the software architecture.

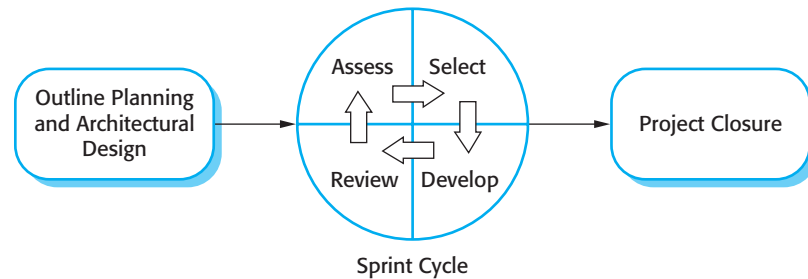


Figure 3.8 The Scrum process

This is followed by a series of sprint cycles, where each cycle develops an increment of the system. Finally, the project closure phase wraps up the project, completes required documentation such as system help frames and user manuals, and assesses the lessons learned from the project.

The innovative feature of Scrum is its central phase, namely the sprint cycles. A Scrum sprint is a planning unit in which the work to be done is assessed, features are selected for development, and the software is implemented. At the end of a sprint, the completed functionality is delivered to stakeholders. Key characteristics of this process are as follows:

1. Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint.
3. The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
4. Once these are agreed, the team organizes themselves to develop the software. Short daily meetings involving all team members are held to review progress and if necessary, reprioritize work. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called ‘Scrum master’. The role of the Scrum master is to protect the development team from external distractions. The way in which the work is done depends on the problem and the team. Unlike XP, Scrum does not make specific suggestions on how to write requirements, test-first development, etc. However, these XP practices can be used if the team thinks they are appropriate.
5. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

The idea behind Scrum is that the whole team should be empowered to make decisions so the term ‘project manager’, has been deliberately avoided. Rather, the

‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog, and communicates with customers and management outside of the team.

The whole team attends the daily meetings, which are sometimes ‘stand-up’ meetings to keep them short and focused. During the meeting, all team members share information, describe their progress since the last meeting, problems that have arisen, and what is planned for the following day. This means that everyone on the team knows what is going on and, if problems arise, can replan short-term work to cope with them. Everyone participates in this short-term planning—there is no top-down direction from the Scrum master.

There are many anecdotal reports of the successful use of Scrum available on the Web. Rising and Janoff (2000) discuss its successful use in a telecommunication software development environment, and they list its advantages as follows:

1. The product is broken down into a set of manageable and understandable chunks.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything and consequently team communication is improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works.
5. Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams with team members located in different places around the world. Consequently, there are various experiments going on to develop Scrum for distributed development environments (Smits and Pshigoda, 2007; Sutherland et al., 2007).

3.5 Scaling agile methods

Agile methods were developed for use by small programming teams who could work together in the same room and communicate informally. Agile methods have therefore been mostly used for the development of small and medium-sized systems. Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to larger systems. Consequently, there has been a great deal of interest in scaling agile methods to cope with larger systems, developed by large organizations.

Denning et al. (2008) argue that the only way to avoid common software engineering problems, such as systems that don't meet customer needs and budget overruns, is to find ways of making agile methods work for large systems. Leffingwell (2007) discusses which agile practices scale to large systems development. Moore and Spens (2008) report on their experience of using an agile approach to develop a large medical system with 300 developers working in geographically distributed teams.

Large software system development is different from small system development in a number of ways:

1. Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.
2. Large systems are 'brownfield systems' (Hopkins and Jenkins, 2008); that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.
3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.
4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

There are two perspectives on the scaling of agile methods:

1. A 'scaling up' perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team.

2. A ‘scaling out’ perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

Agile methods have to be adapted to cope with large systems engineering. Leffingwell (2007) argues that it is essential to maintain the fundamentals of agile methods—flexible planning, frequent system releases, continuous integration, test-driven development, and good team communications. I believe that the critical adaptations that have to be introduced are as follows:

1. For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed and there has to be documentation produced to describe critical aspects of the system, such as database schemas, the work breakdown across teams, etc.
2. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as e-mail, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
3. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. This may mean that new configuration management tools that support multi-team software development have to be introduced.

Small software companies that develop software products have been amongst the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to ‘scale out’ these methods across the organization. Lindvall, et al. (2004) discuss some of the problems in scaling-out agile methods in four large technology companies.

It is difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software

tools (e.g., requirements management tools) and the use of these tools is mandated for all projects.

3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made and this conflicts with the notion of refactoring. In XP, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with the test-first and test-often approaches used in XP.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. They must devote significant resources to the change process. At the time of writing, few large companies have made a successful transition to agile development across the organization.

KEY POINTS

- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads, and producing high-quality code. They involve the customer directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement, and customer participation in the development team.
- A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.

- The Scrum method is an agile method that provides a project management framework. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest-priority tasks for a sprint.
- Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation. Continuous integration is practically impossible when there are several separate development teams working on a project.

FURTHER READING

Extreme Programming Explained. This was the first book on XP and is still, perhaps, the most readable. It explains the approach from the perspective of one of its inventors and his enthusiasm comes through very clearly in the book. (Kent Beck, Addison-Wesley, 2000.)

‘Get Ready for Agile Methods, With Care’. A thoughtful critique of agile methods that discusses their strengths and weaknesses, written by a vastly experienced software engineer. (B. Boehm, *IEEE Computer*, January 2002.) <http://doi.ieeecomputersociety.org/10.1109/2.976920>.

Scaling Software Agility: Best Practices for Large Enterprises. Although focused on issues of scaling agile development, this book also includes a summary of the principal agile methods such as XP, Scrum, and Crystal. (D. Leffingwell, Addison-Wesley, 2007.)

Running an Agile Software Development Project. Most books on agile methods focus on a specific method but this book takes a different approach and discusses how to put XP into practice in a project. Good, practical advice. (M. Holcombe, John Wiley and Sons, 2008.)

EXERCISES

- 3.1. Explain why the rapid delivery and deployment of new systems is often more important to businesses than the detailed functionality of these systems.
- 3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.
- 3.3. When would you recommend *against* the use of an agile method for developing a software system?
- 3.4. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.

- 3.5. Explain why test-first development helps the programmer to develop a better understanding of the system requirements. What are the potential difficulties with test-first development?
- 3.6. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.
- 3.7. Compare and contrast the Scrum approach to project management with conventional plan-based approaches, as discussed in Chapter 23. The comparisons should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.
- 3.8. You are a software manager in a company that develops critical control software for aircraft. You are responsible for the development of a software design support system that supports the translation of software requirements to a formal software specification (discussed in Chapter 13). Comment on the advantages and disadvantages of the following development strategies:
 - a. Collect the requirements for such a system from software engineers and external stakeholders (such as the regulatory certification authority) and develop the system using a plan-driven approach.
 - b. Develop a prototype using a scripting language, such as Ruby or Python, evaluate this prototype with software engineers and other stakeholders, then review the system requirements. Redevelop the final system using Java.
 - c. Develop the system in Java using an agile approach with a user involved in the development team.
- 3.9. It has been suggested that one of the problems of having a user closely involved with a software development team is that they ‘go native’; that is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem and discuss the advantages and disadvantages of each approach.
- 3.10. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using agile methods, which rely on close team working and pair programming. Discuss the difficulties that this new policy might cause and how you might get around these problems.

REFERENCES

- Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.
- Arisholm, E., Gallis, H., Dyba, T. and Sjöberg, D. I. K. (2007). 'Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise'. *IEEE Trans. on Software Eng.*, **33** (2), 65–86.
- Astels, D. (2003). *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall.
- Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer*, **32** (10), 70–8.
- Beck, K. (2000). *extreme Programming explained*. Reading, Mass.: Addison-Wesley.
- Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.
- Cockburn, A. (2001). *Agile Software Development*. Reading, Mass.: Addison-Wesley.
- Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley.
- Cockburn, A. and Williams, L. (2001). 'The costs and benefits of pair programming'. In *Extreme programming examined*. (ed.). Boston: Addison-Wesley.
- Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley.
- Demarco, T. and Boehm, B. (2002). 'The Agile Methods Fray'. *IEEE Computer*, **35** (6), 90–2.
- Denning, P. J., Gunderson, C. and Hayes-Roth, R. (2008). 'Evolutionary System Development'. *Comm. ACM*, **51** (12), 29–31.
- Drobna, J., Noftz, D. and Raghu, R. (2004). 'Piloting XP on Four Mission-Critical Projects'. *IEEE Software*, **21** (6), 70–5.
- Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.
- Hopkins, R. and Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston, Mass.: IBM Press.
- Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall.
- Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.
- Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. and Kahkonen, T. (2004). 'Agile Software Development in Large Organizations'. *IEEE Computer*, **37** (12), 26–34.

- Martin, J. (1981). *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall.
- Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.
- Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. and Quinnan, R. E. (1980). 'The Management of Software Engineering'. *IBM Systems. J.*, **19** (4), 414–77.
- Moore, E. and Spens, J. (2008). 'Scaling Agile: Finding your Agile Tribe'. *Proc. Agile 2008 Conference*, Toronto: IEEE Computer Society. 121–124.
- Palmer, S. R. and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall.
- Parrish, A., Smith, R., Hale, D. and Hale, J. (2004). 'A Field Study of Developer Pairs: Productivity Impacts and Implications'. *IEEE Software*, **21** (5), 76–9.
- Poole, C. and Huisman, J. W. (2001). 'Using Extreme Programming in a Maintenance Environment'. *IEEE Software*, **18** (6), 42–50.
- Rising, L. and Janoff, N. S. (2000). 'The Scrum Software Development Process for Small Teams'. *IEEE Software*, **17** (4), 26–32.
- Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.
- Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall.
- Smits, H. and Pshigoda, G. (2007). 'Implementing Scrum in a Distributed Software Development Organization'. *Agile 2007*, Washington, DC: IEEE Computer Society.
- Stapleton, J. (1997). *DSDM Dynamic Systems Development Method*. Harlow, UK: Addison-Wesley.
- Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.
- Stephens, M. and Rosenberg, D. (2003). *Extreme Programming Refactored*. Berkley, Calif.: Apress.
- Sutherland, J., Viktorov, A., Blount, J. and Puntikov, N. (2007). 'Distributed Scrum: Agile Project Management with Outsourced Development Teams'. *40th Hawaii Int. Conf. on System Sciences*, Hawaii: IEEE Computer Society.
- Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand.
- Williams, L., Kessler, R. R., Cunningham, W. and Jeffries, R. (2000). 'Strengthening the Case for Pair Programming'. *IEEE Software*, **17** (4), 19–25.



6

Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- have been introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems.

Contents

- 6.1** Architectural design decisions
- 6.2** Architectural views
- 6.3** Architectural patterns
- 6.4** Application architectures

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, as shown in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful. While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.

To help you understand what I mean by system architecture, consider Figure 6.1. This shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large*:

1. Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.
2. Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies. I cover architecture in the large in Chapters 18 and 19, where I discuss distributed systems architectures.

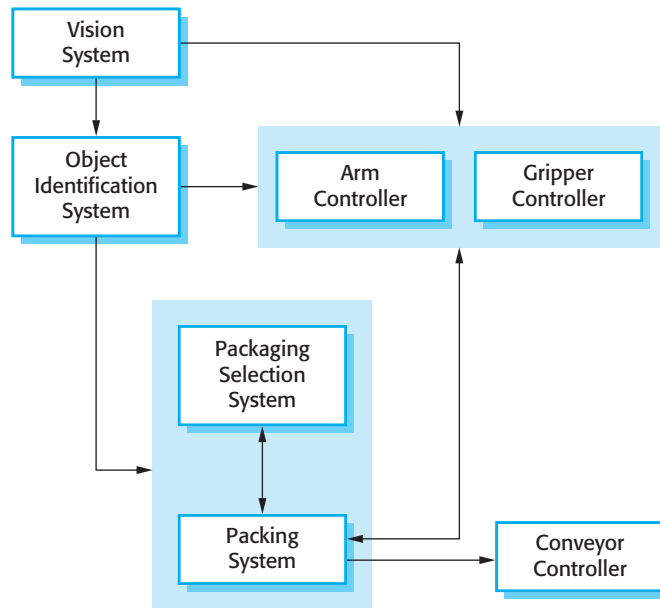


Figure 6.1 The architecture of a packing robot control system

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000). As Bosch discusses, individual components implement the functional system requirements. The non-functional requirements depend on the system architecture—the way in which these components are organized and communicate. In many systems, non-functional requirements are also influenced by individual components, but there is no doubt that the architecture of the system is the dominant influence.

Bass et al. (2003) discuss three advantages of explicitly designing and documenting software architecture:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. *Large-scale reuse* A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

Hofmeister et al. (2000) propose that a software architecture can serve firstly as a design plan for the negotiation of system requirements, and secondly as a means of structuring discussions with clients, developers, and managers. They also suggest that it is an essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

System architectures are often modeled using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's software architecture catalog (Booch, 2009).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. However, in spite of their widespread use, Bass et al. (2003) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. *As a way of facilitating discussion about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.
2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. However, if the architecture of a system is to be thoroughly documented then it is better to use a notation with well-defined semantics for architectural description. However, as I discuss in Section 6.2, some people think that detailed documentation is neither useful, nor really worth the cost of its development.

6.1 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into sub-components?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse. I discuss generic application architectures in Section 6.4 and application product lines in Chapter 16.

For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and

reliability of the system. This is a major topic in its own right and I cover it separately in Chapter 18.

The architecture of a software system may be based on a particular architectural pattern or style. An architectural pattern is a description of a system organization (Garlan and Shaw, 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I discuss a number of frequently used patterns in Section 6.3.

Garlan and Shaw’s notion of an architectural style (style and pattern have come to mean the same thing) covers questions 4 to 6 in the previous list. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on the strategy for decomposing components into sub-components. The approaches that you can use allow different types of architecture to be implemented. Finally, in the control modeling process, you make decisions about how the execution of components is controlled. You develop a general model of the control relationships between the various parts of the system.

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.
2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems in Chapter 13.
5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may

readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, using large components improves performance and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, then some compromise must be found. This can sometimes be achieved by using different architectural patterns or styles for different parts of the system.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic architectural patterns. Bosch's (2000) description of the non-functional characteristics of architectural patterns can also be used to help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation, and for the future evolution of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single architectural model, as each model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact, or the different ways in which system components are distributed across a network. All of these are useful at different times so, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. The views that he suggests are:

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 16) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making. They are a way of communicating the essence of a system to different stakeholders. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but there is no need for a complete description from all perspectives. It may also be possible to associate architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for architectural description (Clements, et al., 2002). A survey in 2006 (Lange et al., 2006) showed that, when the UML was used, it was mostly applied in a loose and informal way. The authors of that paper argued that this was a bad thing. I disagree with this view. The UML was designed for describing object-oriented systems and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description.

I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and which can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers have proposed the use of more specialized architectural description languages (ADLs) (Bass et al., 2003) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because of their specialized nature, domain and application specialists find it hard to understand and use ADLs. This makes it difficult to assess their usefulness for practical software engineering. ADLs designed for a particular domain (e.g., automobile systems) may be used as a

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Figure 6.2 The model-view-controller (MVC) pattern

basis for model-driven development. However, I believe that informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop it. I largely agree with this view and I think that, for most systems, it is not worth developing a detailed architectural description from these four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete. However, an exception to this is when you are developing critical systems, when you need to make a detailed dependability analysis of the system. You may need to convince external regulators that your system conforms to their regulations and so complete architectural documentation may be required.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al., 1995), which prompted the development of other types of pattern, such as patterns for organizational design (Coplien and Harrison, 2004), usability patterns (Usability Group, 1998), interaction (Martin and Sommerville, 2004), configuration management (Berczuk and