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 microsensor 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.

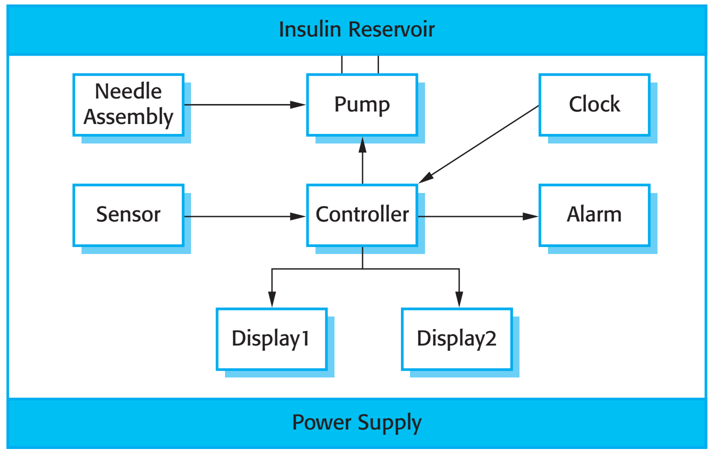



**Figure 1.4** Insulin pump hardware

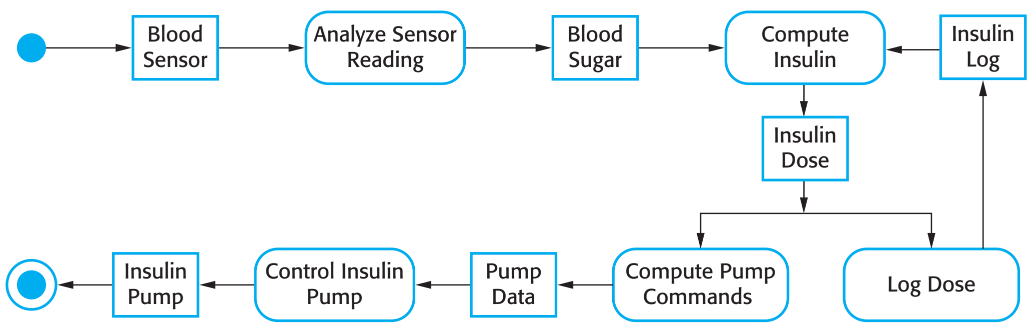



**Figure 1.5** Activity model of the insulin pump

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.

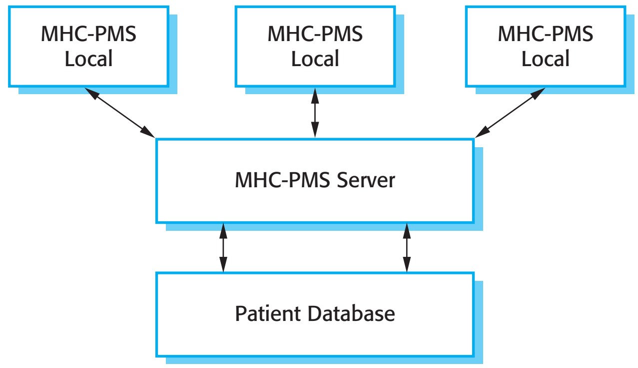



**Figure 1.6** The organization of the MHC-PMS

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

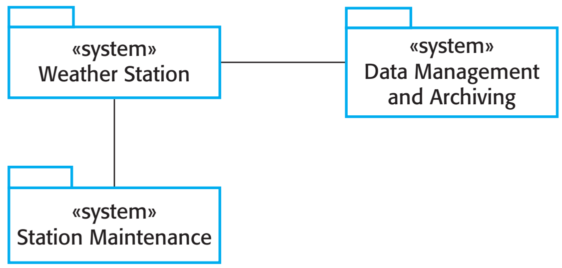



**Figure 1.7** The weather station's environment

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.

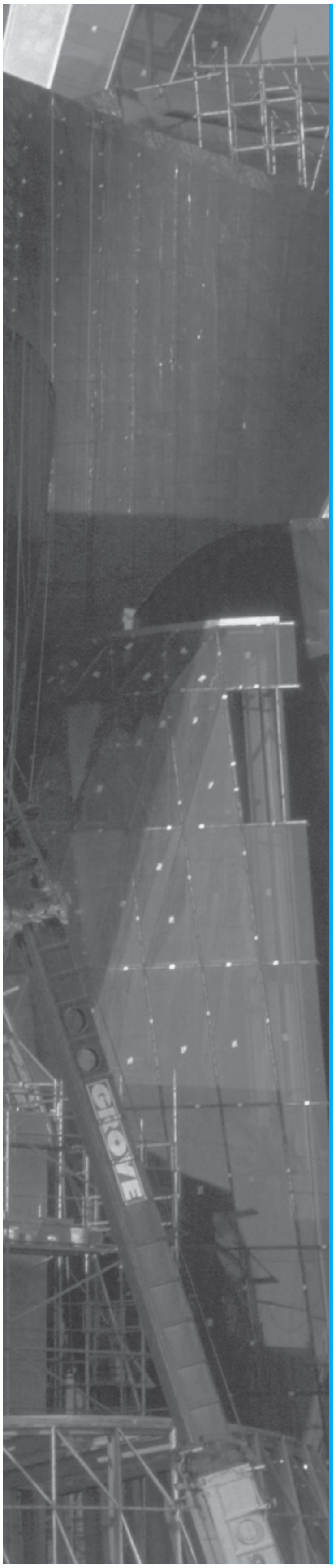
Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Ca.: O'Reilly and Associates.

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

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.

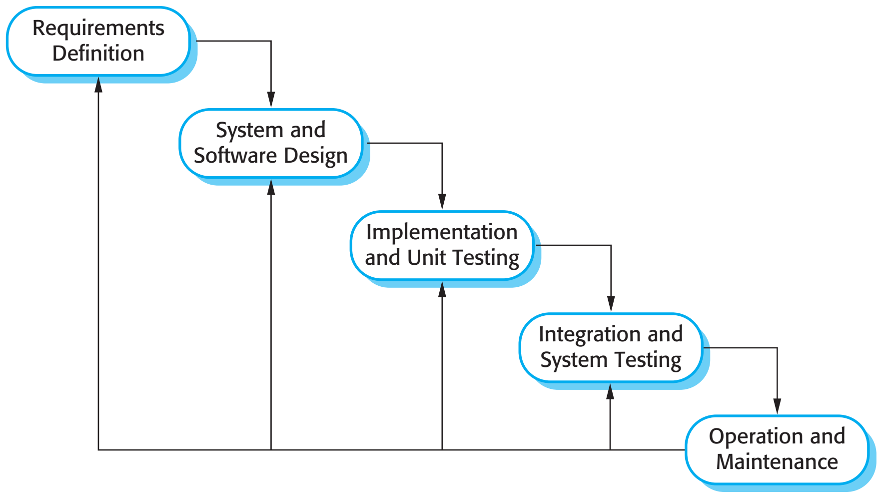



**Figure 2.1** The waterfall model

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

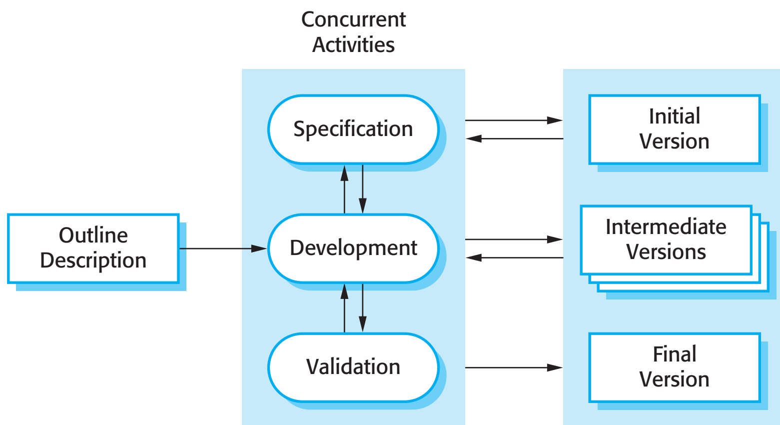



**Figure 2.2** Incremental development

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.

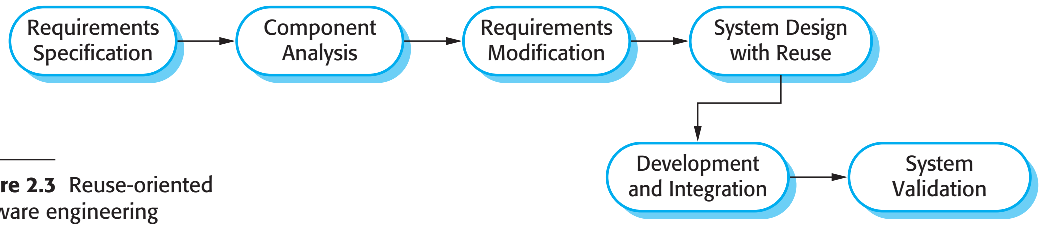



**Figure 2.3** Reuse-oriented software engineering

## 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

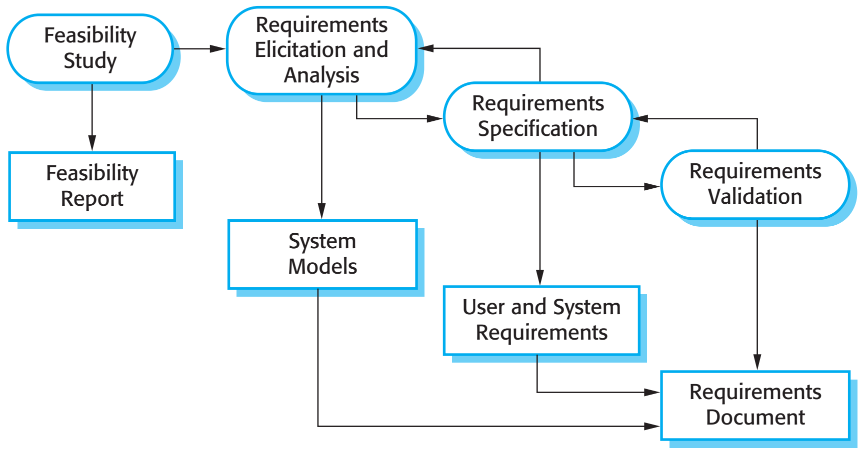



**Figure 2.4** The requirements engineering process

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.

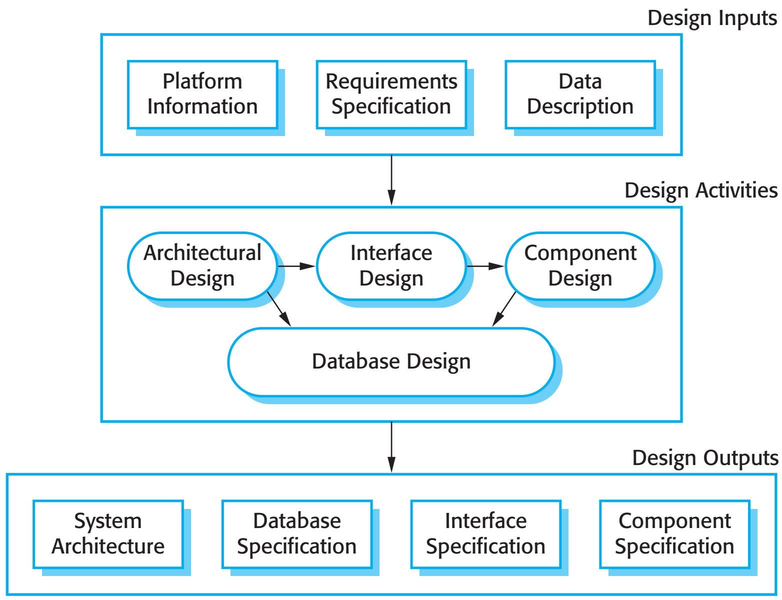



**Figure 2.5** A general model of the design process

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

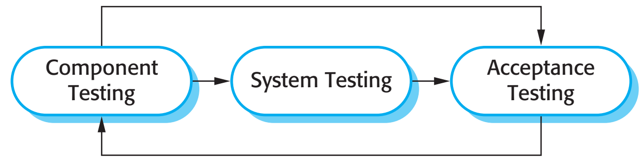



**Figure 2.6** Stages of testing

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