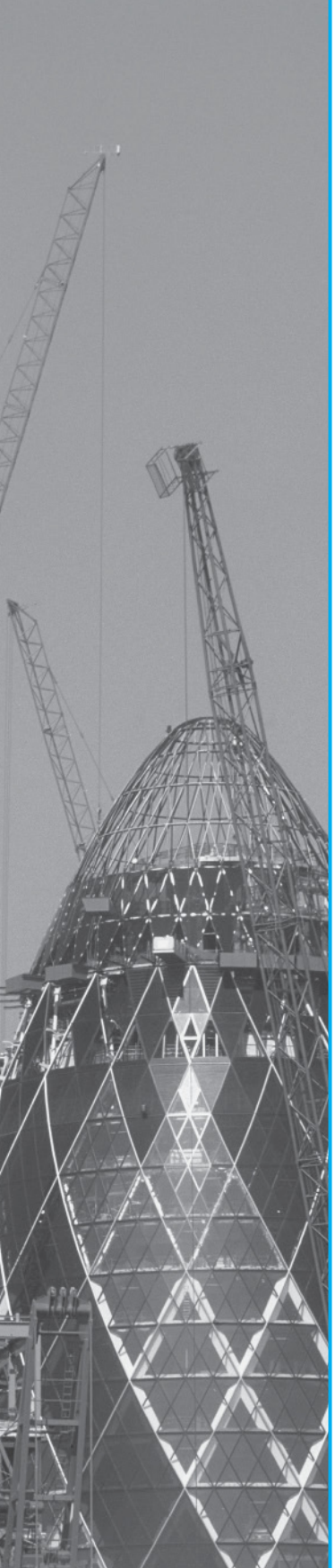

1 Introduction

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 system may require different software engineering techniques;
- understand ethical and professional issues that are important for software engineers;
- have been introduced to four systems, of different types, which are used as examples throughout the book.

Contents

Software engineering is essential for the functioning of government, society, and national and international businesses and institutions. 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. More than 75% of the world's population have a software-controlled mobile phone, and, by 2016, almost all of these will be Internet-enabled.

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they 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 system, ranging from simple embedded systems to complex, worldwide information systems. There are no 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 methods and techniques.

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

1. *Increasing system complexity* 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; and systems have to have new capabilities that were previously thought to be impossible. New software engineering techniques have to be developed to meet new the challenges of delivering more complex software.
2. *Failure to use software engineering methods* It is fairly 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 and we would not have the Internet or modern telecommunications. All forms of travel would be more dangerous and expensive. Challenges for humanity in the 21st century are climate change, fewer natural

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 which are the basis of today's software engineering.

http://software-engineering-book.com/web/history/

resources, changing demographics, and an expanding world population. We will rely on software engineering to develop the systems that we need to cope with these issues.

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; hobbyists write programs for their own interest and enjoyment. However, most software development is a professional activity in which software is developed for business purposes, for inclusion in other devices, or as software products such as information systems and computer-aided design systems. The key distinctions are that professional software is intended for use by someone apart from its developer and that teams rather than individuals usually develop the software. 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 software engineering, I have summarized frequently asked questions about the subject 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, libraries, support websites, and configuration data that are needed to make these programs useful. A professionally developed software system is often more than a single program. A system may consist of several 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

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 from initial conception to operation and maintenance.
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. There are no methods and techniques that are good for everything.
What differences has the Internet made to software engineering?	Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an "app" industry for mobile devices which has changed the economics of software.

Figure 1.1 Frequently asked questions about software engineering

and you don't have to worry about writing program guides, documenting the program design, and so on. 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.

Software engineers are concerned with developing software products, that is, software that can be sold to a customer. There are two kinds of software product:

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 apps for mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This kind of software also includes "vertical"

applications designed for a specific market such as library information systems, accounting systems, or systems for maintaining dental records.

2. *Customized (or bespoke) software* These are systems that are commissioned by and developed for a particular customer. A software contractor designs and implements 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.

The critical distinction between these types of software is that, in generic products, the organization that develops the software controls the software specification. This means that if they run into development problems, they can rethink what is to be developed. For custom products, the specification is 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 systems from SAP and Oracle, 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 consider 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 the software's quality or non-functional 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, an aircraft control system must be safe, 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 think 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 characteristic	Description
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.
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. Software has to be secure so that malicious users cannot 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, resource utilization, etc.
Maintainability	Software should be written in such a way 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.

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 within organizational and financial constraints, and they must 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 development.

Engineering is about getting results of the required quality within 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 the right one for some kinds of software. A more flexible software process that accommodates rapid change is particularly appropriate for the development of interactive web-based systems and mobile apps, which require 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 professional software systems rather than just write programs as

a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance.

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. Four fundamental activities are common to all software processes.

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, as I explain in Chapter 2. 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.

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 are rarely relevant 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.

As I discuss in the next section, there are many different types of software. There are no universal software engineering methods or techniques that may be used. However, there are four related 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 and tablets. 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* Businesses 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 successfully attack our software and that information security is maintained.
4. *Scale* Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

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. The specific methods, tools, and techniques used depend 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 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. However, the SEMAT initiative (Jacobson et al. 2013) proposes that there can be a fundamental meta-process that can be instantiated to create different kinds of process. This is at an early stage of development and may be a basis for improving our current software engineering methods.

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

1. *Stand-alone applications* These are application systems that run on a personal computer or apps that run on a mobile device. They include all necessary functionality and may not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, travel apps, productivity apps, and so on.

2. *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own computers, phones, or tablets. Obviously, these include web applications such as e-commerce applications where you 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 antilock 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 are periodic billing systems, such as phone billing systems, and salary payment systems.

5. *Entertainment systems* These are systems for personal use that are intended to entertain the user. Most of these systems are games of one kind or another, which may run on special-purpose console hardware. 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 and analysis systems* Data collection systems are systems that collect data from their environment and send that data to other systems for processing. The software may have to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location. “Big data” analysis may involve cloud-based systems carrying out statistical analysis and looking for relationships in the collected data.

8. *Systems of systems* These are systems, used in enterprises and other large organizations, that are composed of a number of other software systems. Some of these may be generic software products, such as an ERP system. 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 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 transaction 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.

Each type of system requires specialized software engineering techniques because the software has different characteristics. For example, an embedded control system in an automobile is safety-critical and is burned into ROM (read-only memory) when installed in the vehicle. It is therefore very expensive to change. Such a system needs 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 an interactive web-based system or app, iterative development and delivery is the best approach, 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 systems:

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, the specific process that you should use depends on the type of software that you are developing.
2. Dependability and performance are important for all types of system. 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 effective use 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.

These fundamentals are independent of the program language used for software development. I don't cover specific programming techniques in this book because these vary dramatically from one type of system to another. For example, a dynamic language, such as Ruby, is the right type of language for interactive system development but is inappropriate for embedded systems engineering.

1.1.3 Internet software engineering

The development of the Internet and 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. Wherever it has been possible to do so, businesses have moved to web-based interaction with company software systems.

The notion of software as a service (Chapter 17) was proposed early in the 21st century This has now become the standard approach to the delivery of web-based system products such as Google Apps, Microsoft Office 365, and Adobe Creative Suite. More and more software runs on remote "clouds" instead of local servers and is accessed over the Internet. 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. If you use services such as web-based mail, storage, or video, you are using a cloud-based system.

The advent of the web has led to a dramatic 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 change in software organization has had a major effect on software engineering for web-based systems. 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 preexisting software components and systems, often bundled together in a framework.
2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems are always developed and delivered incrementally.
3. Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services. I discuss this approach to software engineering in Chapter 18.
4. Interface development technology such as AJAX (Holdener 2008) and HTML5 (Freeman 2011) have emerged that support the creation of rich interfaces within a web browser.

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software, as they do to other types of software. Web-based systems are getting larger and larger, so software engineering techniques that deal with scale and complexity are relevant for these systems.

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 regardless 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.

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 (ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version. http://www.acm.org/about/se-code)

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) to extremely serious (dissemination of viruses or other malware).

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, Miller, and Rogerson 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 people involved 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 problems to management? If you discuss these while they are just a suspicion, you may be overreacting to a situation; if you leave it too long, it may be impossible to resolve the difficulties.

We all face such ethical dilemmas 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 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?

[†]ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version Preamble. http://www.acm.org/about/se-code

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 on the views of the people involved. 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 or social media. 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 defense 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, it 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 he or she does not wish to work on such systems, employers should not exert pressure to do so at some later date.

The general area of ethics and professional responsibility is increasingly 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 (Laudon 1995) and Johnson (Johnson 2001). More recent texts such as that by Tavani (Tavani 2013) introduce the notion of cyberethics and cover both the philosophical background and practical and legal issues. They include ethical issues for technology users as well as developers.

I find that a philosophical approach is too abstract and difficult to relate to everyday experience so I prefer the more concrete approach embodied in professional codes of conduct (Bott 2005; Duquenoy 2007). I think that ethics are best discussed in a software engineering context and not as a subject in its own right. Therefore, I do not discuss software engineering ethics in an abstract way but include examples in the exercises that can be the starting point for a group discussion.

1.3 Case studies

To illustrate software engineering concepts, I use examples from four different types of system. I have deliberately not used a single case study, as 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 system types that I use as case studies are:

1. *An embedded system* This is a system where the software controls some hardware device and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, and power management, etc. The example of an embedded system that I use is a software system to control an insulin pump for people who have diabetes.
2. *An information system* The primary purpose of this type of system 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 used is a medical records system.
3. *A sensor-based data collection system* This is a system whose primary purposes are to collect data from a set of sensors and to 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.
4. *A support environment.* This is an integrated collection of software tools that are used to support some kind of activity. Programming environments, such as Eclipse (Vogel 2012) will be the most familiar type of environment for readers of this book. I describe an example here of a digital learning environment that is used to support students' learning in schools.

I introduce each of these systems in this chapter; more information about each of them is available on the website (software-engineering-book.com).

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 that 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 in which the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolizes glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels periodically using an external meter and then estimate the dose of insulin they should inject.

The problem 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. Irregular checking 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,

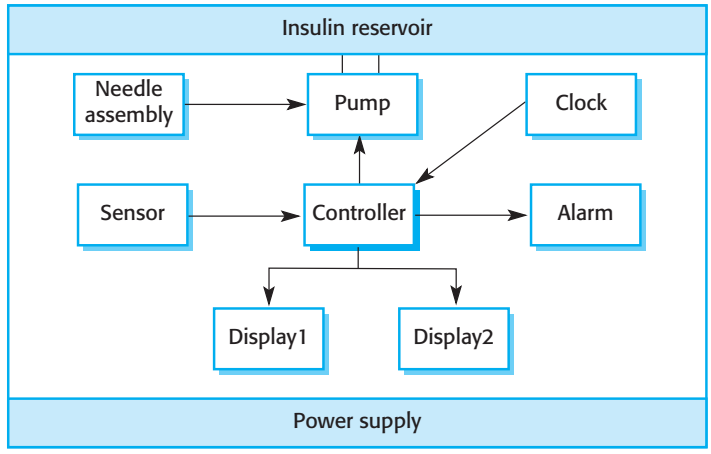

Figure 1.4 Insulin pump hardware architecture

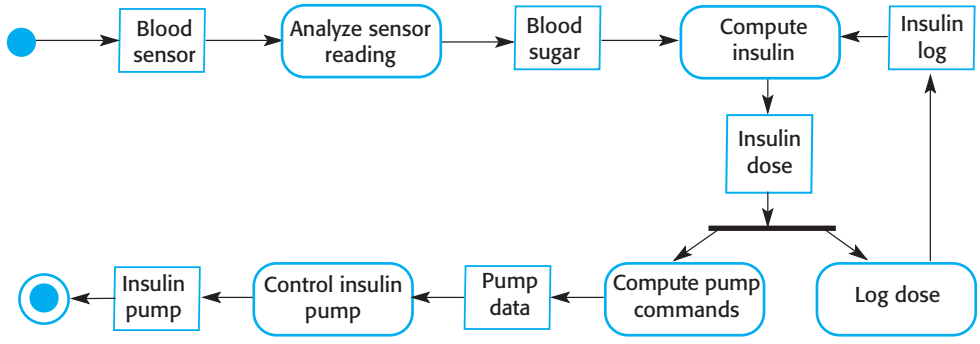

Figure 1.5 Activity model of the insulin pump

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.

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 one are now available and are used by patients who find it difficult to control their insulin levels. In future, it may be possible for diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system uses 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 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 Unified Modeling

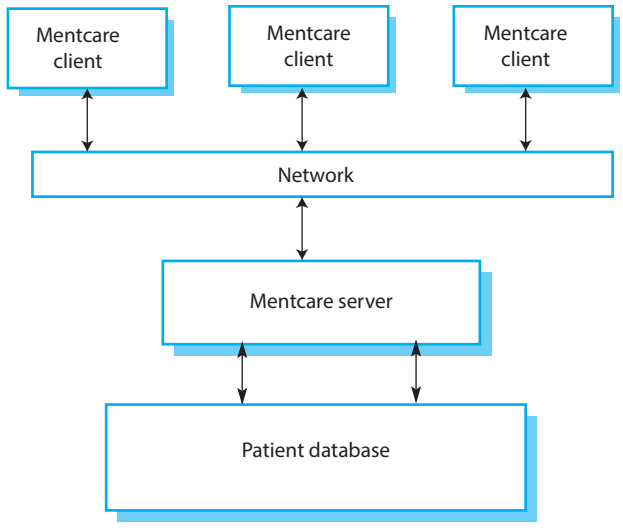

Figure 1.6 The organization of the Mentcare system

Language (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. This system must therefore meet two essential high-level requirements:

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.

The system must therefore be designed and implemented to ensure that it 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 (the Mentcare system) 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 Mentcare system (Figure 1.6) is a patient 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 laptop, 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 and so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems.

This system has two purposes:

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.

Patients who suffer from mental health problems are sometimes irrational and 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"—that is, 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, and so on. 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, the number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system: 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 strict legislative safeguards. One aim of the Mentcare system 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 Mentcare system is also a safety-critical 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 system 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.

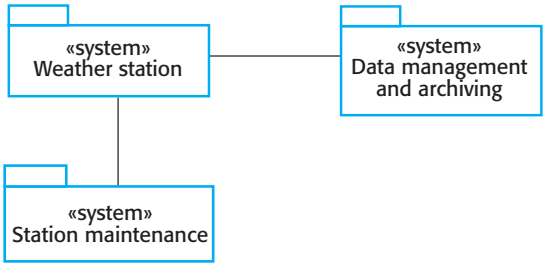

Figure 1.7 The weather station's environment

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 the weather station.

In Figure 1.7, I have used the UML package symbol to indicate that each system is a collection of components and the separate systems are identified 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.

The weather stations include instruments that measure weather parameters such as wind speed and direction, ground and air temperatures, barometric pressure, and 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 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. All communications are through a relatively slow 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.

1.3.4 A digital learning environment for schools

Many teachers argue that using interactive software systems to support education can lead to both improved learner motivation and a deeper level of knowledge and understanding in students. However, there is no general agreement on the 'best' strategy for computer-supported learning, and teachers in practice use a range of different interactive, web-based tools to support learning. The tools used depend on the ages of the learners, their cultural background, their experience with computers, equipment available, and the preferences of the teachers involved.

A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded, plus a set of applications that are geared to the needs of the learners using the system. The framework provides general services such as an authentication service, synchronous and asynchronous communication services, and a storage service.

The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs. These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games, and simulations. They may also include specific content, such as content about the American Civil War and applications to view and annotate that content.

Figure 1.8 is a high-level architectural model of a digital learning environment (iLearn) that was designed for use in schools for students from 3 to 18 years of age. The approach adopted is that this is a distributed system in which all components of the environment are services that can be accessed from anywhere on the Internet. There is no requirement that all of the learning tools are gathered together in one place.

The system is a service-oriented system with all system components considered to be a replaceable service. There are three types of service in the system:

1. *Utility services* that provide basic application-independent functionality and that may be used by other services in the system. Utility services are usually developed or adapted specifically for this system.
2. *Application services* that provide specific applications such as email, conferencing, photo sharing, etc., and access to specific educational content such as scientific films or historical resources. Application services are external services that are either specifically purchased for the system or are available freely over the Internet.

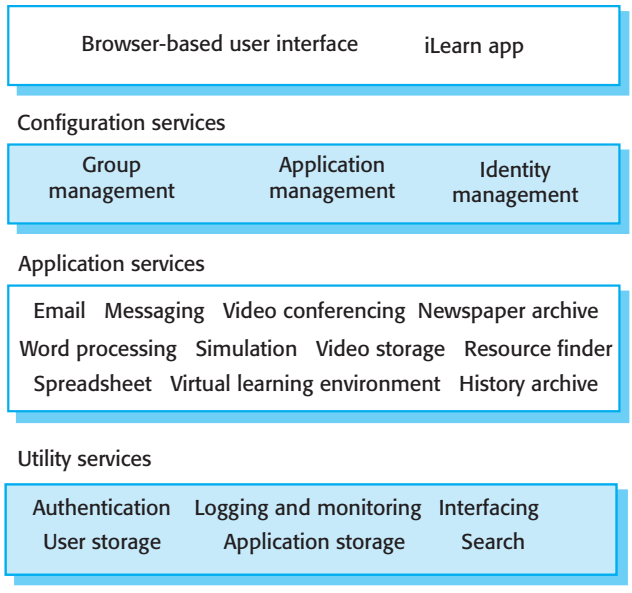

Figure 1.8 The architecture of a digital learning environment (iLearn)

3. *Configuration services* that are used to adapt the environment with a specific set of application services and to define how services are shared between students, teachers, and their parents.

The environment has been designed so that services can be replaced as new services become available and to provide different versions of the system that are suited for the age of the users. This means that the system has to support two levels of service integration:

1. *Integrated services* are services that offer an API (application programming interface) and that can be accessed by other services through that API. Direct service-to-service communication is therefore possible. An authentication service is an example of an integrated service. Rather than use their own authentication mechanisms, an authentication service may be called on by other services to authenticate users. If users are already authenticated, then the authentication service may pass authentication information directly to another service, via an API, with no need for users to reauthenticate themselves.
2. *Independent services* are services that are simply accessed through a browser interface and that operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; reauthentication may be required for each independent service.

If an independent service becomes widely used, the development team may then integrate that service so that it becomes an integrated and supported service.