1 Software and software engineering

The software engineer's job is to solve problems economically by developing high-quality software. In this first chapter we will present important issues that all software engineers should understand to do their jobs well.

In this chapter you will learn about the following

- How does software differ from other products? How does software change over time? What do we mean when we talk about high-quality software? What types of software are there and what are their main differences?
- How are software projects organized? How successful are typical projects?
- How can we define software engineering? Why will following a disciplined approach to software engineering help us produce successful software systems?
- What activities occur in every software project?
- What should we keep in mind as we perform any software engineering activity?

1.1 The nature of software

Similarly to mechanical engineers who design mechanical systems and electrical engineers who design electrical systems, software engineers design software systems. However, software differs in important ways from the types of artifacts produced by other types of engineers:

- Software is largely intangible. Unlike most other engineering artifacts, you cannot feel the shape of a piece of software, and its design can be hard to

Offshoring: an exaggerated fear?

The software engineering labor market has been increasingly affected by the recent trend towards offshoring: this occurs when organizations in developed countries outsource software development to countries that have much lower labor costs yet have highly educated populations and are politically stable. India and some Eastern European countries have particularly benefited from this. Many economists believe offshoring represents a healthy redistribution of wealth that will result, in the longer run, in increased wages and consumer demand in the recipient countries. Citizens of these countries are also becoming big consumers of software, increasing the total market.

However, fear that offshoring will contribute to a lack of jobs is one factor that has caused a sharp drop in university computing enrolments in many developed countries. This fear is exaggerated for three reasons.

First, students studying computing still have a much higher chance of finding a job in their field than students studying most other subjects. Second, as we will learn in this book, close and constant interaction with end-users is essential to the development of quality software; it will always therefore remain important to have a significant part of the development team close to the user. And thirdly, as software development becomes distributed, there will be an increasing need for the disciplined approaches to modeling, requirements, architecture and quality assurance as taught in this book.

visualize. It is therefore difficult for people to assess its quality or to appreciate the amount of work involved in its development. This is one of the reasons why people consistently underestimate the amount of time it takes to develop a software system.

- The mass-production of duplicate pieces of software is trivial. Most other types of engineers are very concerned about the cost of each item in terms of parts and labor to manufacture it. In other words, for tangible objects, the processes following completion of design tend to be the expensive ones. Software, on the other hand, can be duplicated at very little cost by downloading over a network or creating a CD. Almost all the cost of software is therefore in its development, not its manufacturing.

- The software industry is labor intensive. It has become possible to automate many aspects of manufacturing and construction using machinery; therefore, other branches of engineering have been able to produce increasing amounts of product with less labor. However, it would require truly 'intelligent' machines to fully automate software design or programming. Attempts to make steps in this direction have so far met with little success.

- It is all too easy for an inadequately trained software developer to create a piece of software that is difficult to understand and modify. A novice programmer can create a complex system that performs some useful function but is highly disorganized in terms of its design. In other areas of engineering, you can create a poor design too, but the flaws will normally be easier to detect since they will not be buried deep within thousands of pages of source code. For

example, if a civil engineer designed an unsafe bridge, it would normally be easy for inspectors to notice the flaws since they know exactly what to look for in each drawing and calculation. A poorly designed software system will usually at least partly work, but many other types of engineering artifact will not work at all if they are badly designed.

- Software is physically easy to modify; however, because of its complexity it is very difficult to make changes that are correct. People tend to make changes without fully understanding the software. As a side effect of their modifications, new bugs appear.
- Software does not *wear out with use* like other engineering artifacts, but instead its *design deteriorates* as it is changed repeatedly. As mentioned in the previous point, changes tend to introduce new defects; consequently the changed software tends to be worse in terms of design than the original. Over time, the designs of successive versions of software may show significant deterioration to the point where a complete redesign is needed.

Taken together, the above characteristics mean that much existing software is of relatively poor quality and is steadily becoming worse. At the same time, there is strong demand for new and changed software, which customers expect to be of high quality and to be produced rapidly. Therefore, software developers have often not been able to live up to the expectations of their managers and customers – many software projects are either never delivered, or are delivered late and over budget. Furthermore, many software systems that are delivered are never put to use because they have so many problems; others require major modification before they can be used.

This whole situation has been called the *software crisis*, despite the fact that the crisis has been going on for several decades. The term 'crisis' was chosen with the hope that the problems which arose as the software industry expanded would be resolved by implementing improved software engineering methods. Although this sentiment still holds true, we now realize that the difficulties of the software industry are, to some extent, a natural consequence of the complex nature of software, coupled with the laws of economics and the vagaries of human psychology.

It is an objective of this book to teach you how to engineer software so that it meets expectations and doesn't contribute to the crisis. To do that, you will have to learn techniques that allow you to minimize or hide the complexity, and take account of economic and psychological realities.

Types of software and their differences

There are many different types of software. One of the most important distinctions is between *custom* software, *generic* software and *embedded* software.

Custom software is developed to meet the specific needs of a particular customer and tends to be of little use to others (although in some cases

developing custom software might reveal a problem shared by several similar organizations). Much custom software is developed in-house within the same organization that uses it; in other cases, the development is contracted out to consulting companies. Custom software is typically used by only a few people and its success depends on meeting their needs.

Examples of custom software include web sites, air-traffic control systems and software for managing the specialized finances of large organizations.

Generic software, on the other hand, is designed to be sold on the open market, to perform functions that many people need, and to run on general-purpose computers. Requirements are determined largely by market research. There is a tendency in the business world to attempt to use generic software instead of custom software because it can be far cheaper and more reliable. The main difficulty is that it might not fully meet the organization's specific needs. Generic software is often called *Commercial Off-The-Shelf* software (COTS), and it is sometimes also called *shrink-wrapped* software since it is commonly sold in packages wrapped in plastic. Generic software producers hope that they will sell many copies, but their success is at the mercy of market forces.

Examples of generic software include word processors, spreadsheets, compilers, web browsers, operating systems, computer games and accounting packages for small businesses.

Embedded software runs specific hardware devices which are typically sold on the open market. Such devices include washing machines, DVD players, microwave ovens and automobiles. Unlike generic software, users cannot usually replace embedded software or upgrade it without also replacing the hardware. The open-market nature of the hardware devices means that developing embedded software has similarities to developing generic software; however, we place it in a different category due to the distinct processes used to develop it.

Since embedded systems are finding their way into a vast number of consumer and commercial products, they now account for the bulk of software copies in existence. Generic systems, on the other hand, account for most of the software running today on general-purpose computers. Although custom software has fewer copies than either of the other types, it accounts for many more distinct systems and hence is what most developers work on.

It is possible to take generic software and customize it. The risk in doing this, however, is that when a new release of the generic software is issued, the customization work may have to be re-done.

Table 1.1 Differences among custom, generic and embedded software

	Custom	*Generic*	*Embedded*
Number of copies in use	Low	Medium	High
Total processing power devoted to running this type of software	Low	High	Medium
Worldwide annual development effort	High	Medium	Medium

You can also take custom software and try to make it generic; however, this can be a complex process if the software was not designed in a flexible way.

Table 1.1 summarizes some of the important characteristics of custom, generic and embedded software.

Another important way to categorize software in general is whether it is *real-time* or *data processing* software. The most distinctive feature of real-time software is that it has to react immediately (i.e. in real time) to stimuli from the environment (e.g. the pushing of buttons by the user, or a signal from a sensor). Much design effort goes into ensuring that this responsiveness is always guaranteed. Much real-time software is used to operate special-purpose hardware; in fact almost all embedded systems operate in real time. Many aspects of the custom systems that run industrial plants and telephone networks are also real-time.

Generic applications, such as spreadsheets and computer games, have some real-time characteristics, since they must be responsive to their users' inputs. However, these tend to be *soft* real-time characteristics: when timing constraints are not met, such systems merely become sluggish to use. In contrast, most embedded systems have *hard* real-time constraints, and will fail completely if these are not met. Safety is thus a key concern in the design of such systems.

Data processing software is used to run businesses. It performs functions such as recording sales, managing accounts, printing bills etc. The biggest design issues are how to organize the data and provide useful information to the users so they can perform their work effectively. Accuracy and security of the data are important concerns, as is the privacy of the information gathered about people. A key characteristic of traditional data processing tasks is that rather than processing data the moment it is available, it is instead gathered together in batches to be processed at a later time.

Some software has both real-time and data processing aspects. For example, a telephone system has to manage phone calls in real time, but billing for those calls is a data processing activity.

Software varies in terms of its age. Much custom software written in the 1960s and 1970s is still in use today. That software differs from newly developed software in terms of programming languages, data storage technologies, user interface technology and design techniques. Many of the web-based user interfaces we use today, e.g. for banking, are just new *front ends* on much older custom data processing software.

Usage of the word 'software' – a common mistake made by non-native speakers of English.

Many non-native speakers of English erroneously say sentences such as the following: 'I will create *a software* to update the database'. The error is that you cannot talk about 'a software'. When the word 'software' is used as a noun, it is a mass noun, like 'water' and 'sand', and cannot be preceded by the indefinite article 'a'. Therefore you have to say, 'I will create *some software* to update the database', or 'I will create *a piece of software* to update the database'. You can also use the word software as an adjective, as in 'I will create *a software system* to update the database'. In this latter case the indefinite article is referring to 'system', not 'software'.

Exercise

E1 Classify the following software according to whether it is likely to be *custom*, *generic* or *embedded (or some combination)*; and whether it is *data processing* or *real-time*.

(a) A system to control the reaction rate in a nuclear reactor.

(b) A program that runs inside badges worn by nuclear plant workers that monitors radiation exposure.

(c) A program used by administrative assistants at the nuclear plant to write letters.

(d) A system that logs all activities of the reactor and its employees so that investigators can later uncover the cause of any accident.

(e) A program used to generate annual summaries of the radiation exposure experienced by workers.

(f) An educational web site containing a Flash animation describing how the nuclear plant works.

1.2 What is software engineering?

Not all software development should be called software engineering, in the same way as not all construction is civil engineering. A do-it-yourselfer can build a wooden footbridge spanning a 60-cm-wide stream in his or her garden, but it requires a civil engineer to build a bridge across a wider span that public vehicles will traverse. Similarly, a self-trained shareware author may write a small program to track a personal stock portfolio, but it requires a software engineer to develop a complete trading and accounting system for a large brokerage company.

Definition: *software engineering* is the process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

Each of the words in this definition has been chosen carefully. Let us therefore split up the definition and examine each component.

Solving customers' problems

Solving customers' problems should be the *goal* of every software engineering project. Before finalizing any software engineering decision, you should therefore ask yourself whether the proposed alternative will help achieve this goal. In particular, it is important to recognize activities that are not consistent

with this goal, such as adding unnecessary features. Software engineers have the responsibility to recognize situations when it would be most cost effective *not* to develop software at all, to develop *simpler* software or to purchase *existing* software.

The problems being solved by software engineers are usually related to human activities. Software engineers must therefore learn to communicate and negotiate effectively with people, to understand how people do their work, and to understand what impact any proposed software may have on its users' productivity.

Systematic development and evolution

Software development becomes an engineering process when the developers apply well-understood techniques in an organized and disciplined way. Software engineering is a young field, and its technology and techniques are still undergoing rapid development. Nevertheless, there are many well-accepted practices that have been formally standardized by bodies such as the IEEE, ISO (International Organization for Standardization) and various national standards bodies.

Sometimes a software engineering team sets out to develop completely new software. However, most development work involves modifying software that has been already written – this is because software is normally continually changed over a period of years until it becomes obsolete. Ensuring that this constant change, called *maintenance* or *evolution*, is done in a systematic way is an integral part of software engineering. We will discuss this in more detail in Section 1.6 below.

Large, high-quality software systems

A small system can often be successfully developed by a programmer working alone. However, large systems with many functions and components become too complex unless engineering discipline is applied. A system of many thousands of lines of code cannot be completely understood by one person, and certainly would take one person far too long to develop, therefore teamwork is essential to software engineering. One of the hardest challenges is dividing up the work and ensuring that the teams communicate effectively and produce subsystems that properly connect with each other to produce a large but functioning system.

The techniques discussed in this book are therefore *essential* for large systems, although many of them are also *useful* for small systems.

The end product that is produced must be of sufficient quality. Some software engineering techniques are aimed at increasing the quality of the design, whereas others are used to verify that sufficient quality is present before the software is released. Quality is discussed in more detail in Section 1.5 and Chapter 10.

Cost, time and other constraints

One of the essential characteristics of engineering is that you have to consider economic constraints as you try to solve each problem. The main economic constraints are: 1) resources are finite, 2) it is not worth doing something unless the benefit gained from it outweighs its cost, and 3) if somebody else can perform some particular task more cheaply or faster than us, they will probably succeed instead of us. Software engineers, like other engineers, therefore must ensure their systems can be produced within a limited budget and by a certain due date. Achieving this requires careful planning and sticking to the plan in a disciplined way. Furthermore, creating a realistic plan in the first place requires a great deal of knowledge about what is required to produce a system, and how long each activity should take.

Unfortunately, failure to stick to cost and time budgets has been widespread in software engineering projects. The reasons for this are many, but include the inherent complexity of software, the relative immaturity of software engineering and its technologies, lack of knowledge and experience on the part of software engineers, the inherent human tendency towards over-confidence, and pressure to offer excessively low prices and short development times in order to obtain contracts or make sales.

Other definitions of software engineering

We have presented our definition of software engineering. Here are two other definitions:

- IEEE: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).
- The Canadian Standards Association: The systematic activities involved in the design, implementation and testing of software to optimize its production and support.

1.3 Software engineering as a branch of the engineering profession

People have talked about software engineering since 1968 when the term was coined at a NATO conference. However, only since the mid-1990s has there been a shift towards recognizing software engineering as a distinct branch of the engineering profession. Some parts of the world, notably Europe and Australia, were somewhat ahead of others in this regard.

In most countries, in order to legally perform consulting or self-employed work where you call yourself an 'engineer', you must be licensed. Similarly, a company that sells engineering services may be required to employ licensed engineers who take formal responsibility for projects, ensuring they are conducted following accepted engineering practices.

Prior to the 1940s, very few jurisdictions required engineers to be licensed. However, various disasters caused by the failure of designs eventually convinced almost all governments to establish licensing requirements. Licensing agencies have the responsibility to ensure that anyone who calls himself or herself an engineer has sufficient engineering education and experience. To exercise this responsibility, the agencies accredit educational institutions they believe are providing a proper engineering education, and

Ethics in Software Engineering

It is very important as a software engineer-in-training that you develop a sense of professional ethics. Many people perform software development work without fully realizing some of the ethical issues that can arise. The following are highlights of the IEEE/ACM code of ethics. For details about the IEEE and the ACM, see the 'For More Information' section at the end of the chapter.

Software engineers shall:

- Act consistently with the public interest.
- Act in the best interests of their client or employer, as long as this is consistent with the public interest.
- Develop and maintain their product to the highest standards possible.
- Maintain integrity and independence when making professional judgments.
- Promote an ethical approach in management.
- Advance the integrity and reputation of the profession, as long as doing so is consistent with the public interest.
- Be fair and supportive to colleagues.
- Participate in lifelong learning.

scrutinize the background of those who are applying to be engineers, often requiring them to write exams.

We can characterize the work of engineers as follows: engineers *design* artifacts following well-accepted practices, which normally involve the application of science, mathematics and economics. Since engineering has become a licensed profession, adherence to *codes of ethics* and taking *personal responsibility* for work have also become essential characteristics. Some people only include in engineering those design activities that have a potential to impact public safety and well-being; however, since most people who are trained as engineers do not in fact work on such critical projects, most people define engineering in the broader sense.

Historically, engineering has evolved several specialties, most notably civil, mechanical, electrical and chemical engineering. *Computer engineering* evolved in the 1980s to focus on the design of computer systems that involve both hardware and software components. However, most of the practitioners performing what we have defined above to be software engineering have not historically been formally educated as engineers.

Many of the earliest programmers were mathematicians or physicists; then in the 1970s the discipline of *computer science* developed, and educated many of the current generation of software developers. The computer science community recognized the need for a disciplined approach to the creation of large software systems, and developed the software engineering discipline.

In the mid-1990s the first jurisdictions started to recognize software engineering as a distinct branch of engineering. For example, in the United Kingdom those who study software engineering in computer science departments

at universities have been able to achieve the status of Chartered Engineer, after a standard period of work experience and passing certain exams. In North America, the State of Texas and the Province of Ontario were among the first jurisdictions to license software engineers (in 1998 and 1999 respectively).

In parallel with the process of licensing software engineers, universities have been establishing academic programs in universities that focus on software engineering, and are clearly distinct from either computer science or computer engineering. Since considerable numbers of these graduates are now entering the workforce, software engineering has become firmly established as a branch of engineering.

1.4 Stakeholders in software engineering

Many people are involved in a software engineering project and expect to benefit from its success. We will classify these *stakeholders* into four major categories, or roles, each having different motivations, and seeing the software engineering process somewhat differently.

- **Users**. These are the people who will use the software. Their goals usually include doing enjoyable or interesting work, and gaining recognition for the work they have done. Often they will welcome new or improved software, although some might fear it could jeopardize their jobs. Users appreciate software that is easy to learn and use, makes their life easier, helps them achieve more, or allows them to have fun.
- **Customers** (also known as *clients*). These are the people who make the decisions about ordering and paying for the software. They may or may not be users – the users may work for them. Their goal is either to increase profits or simply to run their business more effectively. Customers appreciate software that helps their organization save or make money, typically by improving the productivity of the users and the organization as a whole. If you are developing custom software, then you know who your customers are; if you are developing generic software, then you often only have *potential* customers in mind.
- **Software developers**. These are the people who develop and maintain the software, many of whom may be called software engineers. Within the development team there are often specialized roles, including requirements specialists, database specialists, technical writers, configuration management specialists, etc. Development team members normally desire rewarding careers, although some are more motivated by the challenge of solving difficult problems or by being a well-respected 'guru' in a certain area of expertise. Many developers are motivated by the recognition they receive by doing high-quality work.
- **Development managers**. These are the people who run the organization that is developing the software; they often have an educational background in

business administration. Their goal is to please the customer or sell the most software, while spending the least money. It is important that they have considerable knowledge about how to manage software projects, but they may not be as intimately familiar with small details of the project as are some of the software developers. For this reason, it is important that software developers keep their managers informed of any problems.

In some cases, two, three or even all four of these stakeholder roles may be held by the same person. In the simplest case, if you were privately developing software for your own use, then you would have all four roles.

Exercise

E2 How do you think each of the four types of stakeholders described above would react in each of the following situations?

(a) You study a proposal for a new system that will completely automate the work of one individual in the customer's company. You discover that the cost of developing the system would be far more than the cost of continuing to do the work manually, so you recommend against proceeding with the project.

(b) You implement a system according to the precise specifications of a customer. However, when the software is put into use, the users find it does not solve their problem.

1.5 Software quality

Almost everybody says they want software to be of 'high quality'. But what does the word 'quality' really mean? There is no single answer to this question since, like beauty, quality is largely in the eye of the beholder.

Figure 1.1 shows what quality means to each of the stakeholders. They each consider the software to be of good quality if the outcome of its development and maintenance helps them meet their personal objectives.

Attributes of software quality

The following are five of the most important attributes of software quality. Software engineers try to balance the relative importance of these attributes so as to design systems with the best overall quality, as limited by the money and time available.

- **Usability**. The higher the usability of software, the easier it is for users to work with it. There are several aspects of usability, including learnability for novices, efficiency of use for experts, and handling of errors. We will discuss more about usability in Chapter 7.

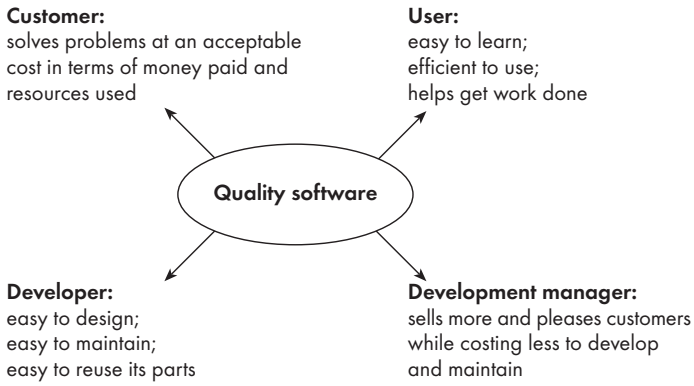

Figure 1.1 What software quality means to different stakeholders

- **Efficiency**. The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth and other resources. This is important to customers in order to reduce their costs of running the software, although with today's powerful computers, CPU-time, memory and disk usage are less of a concern than in years gone by.
- **Reliability**. Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily.
- **Maintainability**. This is the ease with which you can change the software. The more difficult it is to make a change, the lower the maintainability. Software engineers can design highly maintainable software by anticipating future changes and adding flexibility. Software that is more maintainable can result in reduced costs for both developers and customers.
- **Reusability**. A software component is reusable if it can be used in several different systems with little or no modification. High reusability can reduce the long-term costs faced by the development team. We will discuss reusable technology in Chapter 3.

All of these attributes of quality are important. However, the relative importance of each will vary from stakeholder to stakeholder and from system to system. For example, reliability and efficiency are usually both of concern to customers and users; however, in a safety-critical system for controlling a nuclear power plant, reliability would be far more important than efficiency – assuming that faster hardware could be bought if efficiency became a problem. On the other hand, efficiency might be highly important in a program for biologists that calculates how proteins fold – such a program might take days to run, but if it fails no disaster will occur. The program can simply be corrected and re-run.

Often, software engineers improve one quality at the expense of another. In other words, they have to consider various *trade-offs*. The following are some examples of this:

- Improving efficiency may make a design less easy to understand. This can reduce maintainability, which leads to defects that reduce reliability.
- Achieving high reliability often entails repeatedly checking for errors and adding redundant computations; achieving high efficiency, in contrast, may require removing such checks and redundancy.
- Improving usability may require adding extra code to provide feedback to the users, which might in turn reduce overall efficiency and maintainability.

One of the characteristics that distinguishes good engineering practice is setting *objectives* for quality when starting a project, and then designing the system to meet these objectives. The objectives are set in such a way that if they are met, all the stakeholders will be happy. Also, since there is no need to exceed the objectives, they help engineers to avoid spending more effort than is necessary.

To compete in the market successfully, it is sometimes necessary to *optimize* certain aspects of designs. This means achieving the best possible levels of certain qualities, while not exceeding a certain budget and at the same time meeting objectives for the other qualities.

Exercise

E3 For each of the following systems, which attributes of quality do you think would be the most important and the least important?

(a) A web-based banking system, enabling the user to do all aspects of banking on-line.

(b) An air traffic control system.

(c) A program that will enable users to view digital images or movies stored in all known formats.

(d) A system to manage the work schedule of nurses that respects all the constraints and regulations in force at a particular hospital.

(e) An application that allows you to purchase any item seen while watching TV.

Internal quality criteria

Above, we have largely been talking about *external quality attributes* that can be observed by the stakeholders and have a direct impact on them. There are also many *internal quality criteria* that characterize aspects of the design of software

and have an effect on the external quality attributes. The following are a couple of examples:

- The amount of commenting of the code. This can be measured as the fraction of total lines in the source code that are comments. This impacts maintainability, and indirectly it impacts reliability.
- The complexity of the code measured in terms of the nesting depth, the number of branches and the use of certain complex programming constructs. This directly impacts maintainability and reliability.

In Sections 2.10 and 9.2, when we talk about design, we will discuss additional internal quality criteria that affect the externally visible qualities.

Quality for the short term vs. quality for the long term

It is human nature to worry more about short-term needs and ignore the longer-term consequences of decisions. This can have severe consequences. Examples of short-term quality concerns are: Does the software meet the customer's immediate needs? Is it sufficiently efficient for the volume of data we have today?

These questions are important, and must be answered. However, if you take an exclusively short-term focus you are likely to ignore maintainability, and also to ignore the longer-term needs of the customers. This is a mistake made by numerous software engineers over the years, resulting in much higher costs later on. Unfortunately, at the height of excitement about new projects with impending deadlines and markets to capture, even seasoned developers fall into the same trap.

1.6 Software engineering projects

Software engineering work is normally organized into projects. For a small software system, there may only be a single team of three or four developers working on the project. For a larger system, the work is usually subdivided into many smaller projects.

We can divide software projects into three major categories: 1) those that involve modifying an existing system; 2) those that involve starting to develop a system from scratch, and 3) those that involve building most of a new system from existing components, while developing new software only for missing details.

Evolutionary projects

Most software projects are of the first type – modifying an existing system. The term *maintenance* is often used to describe this process; however, for many people the word maintenance implies keeping something running by simply fixing problems, but without adding significant new features. The reality of

software change is somewhat different: there tends to be constant pressure from users and customers not only to fix problems but also to make many other kinds of changes. After several years of such changes, software systems are often significantly larger and barely resemble their original state. We will thus use the term *evolution* to more accurately describe what happens to software over its life-span.

Evolutionary or maintenance projects can be of several different types:

- *Corrective* projects involve fixing defects.
- *Adaptive* projects involve changing the system in response to changes in the environment in which the software runs. For example, it might be necessary to make changes so that the system will continue to work with a new version of the operating system or database, or with a new set of tax laws.
- *Enhancement* projects involve adding new features for the users.
- *Re-engineering* or *perfective* projects involve changing the system internally so that it is more maintainable, without making significant changes that the user will notice.

In reality, most evolutionary projects involve more than one of the above.

In many cases, a software engineering team must undertake evolution of a system when the original developers are no longer available, or when their memory of the design is starting to fade. Such a system is called a *legacy* system.

A team can take great pride in evolving a high-quality product such that it continues to meet the needs of customers. However, it is important to ensure that the product does not become a 'victim of its own success'. This occurs when customers constantly want new features added, so the software becomes so large and bloated that it becomes difficult to maintain at a high level of quality.

Greenfield projects

Projects to develop an entirely new software system from scratch are significantly less common than evolutionary projects. Developers often enjoy such brand new, or *greenfield*, projects because they have a wider freedom to be creative about the design.

In a greenfield project you are not constrained by the design decisions and errors made by predecessors. However, it takes a lot of work to build a complex system from scratch.

Projects that involve building on a framework or a set of existing components

The third type of software project can be considered neither evolutionary nor new development. This type of project, which is becoming increasingly common, starts with a *framework*, or involves plugging together several *components* that are already developed and provide significant functionality.

A framework is a software system especially designed to be reused in different projects, or in the various products of a *product line*. A framework contains important functionality, but must be adapted to handle the requirements of particular customers or products.

For example, imagine an application framework for ticketing. Such a system would have basic capabilities for reserving and printing tickets for events or travel. These functions would be well designed and tested by the original developers of the framework. However, many details would need to be added to handle the particular needs of each new organization that adopts the framework. Selling tickets for a theater can be quite different from selling tickets for a sporting event, a tropical holiday package or even a cinema.

As an example of the use of components, imagine you had an accounting package and a package for tracking meetings, appointments etc. You might hook these together to create a product for a lawyer's office. The meetings and appointments would automatically result in charges for time being recorded in the accounting package. The code that you write to connect the two component packages is called *glue*.

The use of frameworks or components allows you to benefit from reusing software that has been shown to be reliable. Yet, at the same time, it gives you much of the freedom to innovate that you would have if you were performing greenfield development.

In Chapter 3 we will discuss frameworks in detail. We will also present a framework that you will use in exercises and projects throughout this book.

1.7 Activities common to software projects

The following subsections briefly describe many of the activities commonly found in software engineering projects. We will discuss most of these in more detail later in the book.

Requirements and specification

In order to solve the customer's problems, you must first understand the problems, the customer's business environment, and the available technology which can be used to solve the problems. Once you have done this, you can meet with the customers and users to decide on a course of action that will solve the problems. If you decide that developing or modifying software is the best course of action, then you can decide in detail what facilities the software should provide.

This overall process may include the following activities.

- **Domain analysis:** understanding the background needed so as to be able to understand the problem and make intelligent decisions.
- **Defining the problem:** narrowing down the scope of the system by determining the precise problem that needs solving.

- **Requirements gathering:** obtaining all the ideas people have about what the software should do.
- **Requirements analysis:** organizing the information that has been gathered, and making decisions about what in fact the software should do. The term 'requirements analysis' is often used more broadly to include some of the other steps in this list.
- **Requirements specification:** writing a *precise* set of instructions that define what the software should do. These instructions should describe how the software behaves from the perspective of the user, but should not describe any details of the implementation.

  One of the most important principles of requirements is to separate the 'what' from the 'how'. The 'what' refers to the requirements – what is needed to solve the problem. The 'how' refers to how the solution will be designed and implemented.

  Although initial requirements should be established early in a project, the customers' needs tend to change. Requirements analysis therefore should be continued throughout the life of a software system. We will discuss requirements in detail in Chapter 4.

Design

Design is the process of deciding how the requirements should be implemented using the available technology. Important activities during design include:

- Deciding what requirements should be implemented in hardware and what in software. This is called *systems engineering* and is normally only necessary for embedded and other real-time systems. Even for these systems, there is a trend towards implementing more and more facilities in software so that the hardware can be simpler and more generic.
- Deciding how the software is to be divided into subsystems and how the subsystems are to interact. This process is often called *software architecture*; there are several well-known ways of structuring software which are called *architectural patterns* or *styles*. In Chapter 3 we will introduce the client–server architecture, and in Chapter 9 we will look at other architectural patterns.
- Deciding how to construct the details of each subsystem. Such details include the data structures, classes, algorithms and procedures. This process is often called *detailed design*.
- Deciding in detail how the user is to interact with the system, and the *look and feel* of the system. This is called *user interface design*, and will be discussed in Chapter 7.
- Deciding how the data will be stored on disk in databases or files. We do not discuss this topic in this book – it is addressed in many specialized books.

Agile versus conventional development

There is a community of software engineers who practice what is called *agile* development. Agile methods emphasize the ability to quickly modify software and have been found to work well for small to medium-sized systems. The most well-known such method is called eXtreme Programming (XP). We will contrast agile methods with more conventional methods at several places in this book.

One way in which agile and conventional methods differ is in how they treat requirements and design. Agile practitioners gather requirements in very small increments, and design and implement each increment before gathering the next small requirements increment. They fully acknowledge that this may require the design to be changed to accommodate the new requirements, and use techniques called *refactoring* to make the necessary design changes. Conventional practitioners, on the other hand, prefer to develop a design that will be robust in the face of changing requirements. We will revisit all these ideas at various points in the book.

Quite often, for large systems, software engineers work on architectural design in conjunction with high-level requirements. This allows them to divide a system effectively into subsystems. Detailed requirements can then be developed for each subsystem. For smaller systems and lower-level subsystems though, it is conventional to develop the requirements before starting the design since otherwise the design may have to be re-done if requirements change.

Modeling

Modeling is the process of creating a representation of the domain or the software. Various modeling approaches can be used during both requirements analysis and design. These include:

- **Use case modeling**. This involves representing the sequences of actions performed by the users of the software. We will discuss this in Chapter 4.
- **Structural modeling**. This involves representing such things as the classes and objects present in the domain or in the software. This is the topic of Chapters 5 and 6.
- **Dynamic and behavioral modeling**. This involves representing such things as the states that the system can be in, the activities it can perform, and how its components interact. This is the topic of Chapter 8.

Modeling can be performed visually, using diagrams, or else using *semi-formal* or *formal languages* that express the information systematically or mathematically. In this book, we will primarily use semi-formal notations and diagrams – in particular a visual language called *UML*.

Programming

Programming is an integral part of software engineering. It involves the translation of higher-level designs into particular programming languages. It

Pair programming

One of the recommended approaches in the agile method 'eXtreme Programming' is called *pair programming*. In this technique, two programmers always work together in front of a single computer. The idea is that their constant interaction should stimulate good ideas and prevent errors. Whether this approach should be widely adopted is still being studied and debated.

should be thought of as the final stage of design because it involves making decisions about the appropriate use of programming language constructs, variable declarations etc. Most people who call themselves programmers also perform many higher-level design activities. People who limit their work to programming (i.e. who do no higher-level design or analysis) are often today called 'coders'.

One of the objectives of software engineering researchers has been to automate programming. There has been some success in this regard – some tools now generate much of the code for you from models typically represented in UML. However, there will always be a need for some programming done by humans.

We assume that readers of this book have some object-oriented programming background. We will use Java for the example code in this book, and you will be asked to translate designs into programs so you can get a feel for the effects of various design decisions. If you know an object-oriented language other than Java (e.g. C++, C# or Smalltalk) it should not be difficult to learn enough Java to use the book effectively.

Quality assurance

Quality assurance (QA) encompasses all the processes needed to ensure that the quality objectives discussed in Section 1.5 are met. Quality assurance occurs throughout a project, and includes many activities, including the following:

- **Reviews and inspections**. These are formal meetings organized to discuss requirements, designs or code to see if they are satisfactory.
- **Testing**. This is the process of systematically executing the software to see if it behaves as expected.

Quality assurance is also often divided into validation, which is the process of determining whether the requirements will solve the customer's problem, and verification, which is the process of making sure the requirements have been adhered to.

In various chapters, we present checklists that you can use to conduct reviews. Testing and some other aspects of quality assurance are presented in detail in Chapter 10.

Deployment

Deployment involves distributing and installing the software and any other components of the system such as databases, special hardware etc. It also involves managing the transition from any previous system.

Deploying a new release of a large system with many users can pose great difficulties – the amount of work is often under-estimated. To keep this book short, we have decided not to discuss deployment.

Managing software configurations

Configuration management involves identifying all the components that compose a software system, including files containing requirements, designs and source code. It also involves keeping track of these as they change, and ensuring that changes are performed in an organized way. All software engineers must participate in the configuration management of the parts of the system for which they are responsible.

Managing the process

Managing software projects is considered an integral part of software engineering. All software engineers assist their managers to some extent, and most will, at some point in their careers, become managers themselves.

Management issues are discussed briefly in Chapter 11. In addition to leading the other activities described above, the manager has to undertake the following tasks:

- **Estimating the cost of the system**. This involves studying the requirements and determining how much effort they will take to design and implement.
- **Planning**. This is the process of allocating work to particular developers, and setting a schedule with deadlines.

Both cost estimates and plans need to be examined and revised on a regular basis, since initial estimates will only be rough.

1.8 The themes emphasized in this book

The nine general themes discussed below are emphasized through many of the chapters in this book. They represent general principles or unifying approaches that can be used in any software project.

Theme 1: understanding the customer and user

Interaction with customers and users should occur in virtually all of the software engineering activities discussed in the previous section. These two groups of stakeholders are most heavily involved in requirements analysis, user interface design and deployment, but also may play a role in design, quality assurance and project management.

If software engineers can learn how users and customers think and behave, then it will be easier to produce software that meets their needs. Ensuring that they feel involved in the software engineering process will result in fewer mistakes being made and greater acceptance of the finished product.

Theme 2: basing development on solid principles and reusable technology

A fundamental tenet of engineering is that once techniques or technology become well established, their use should become routine. Civil engineers, for example, have a well-established set of principles, which they use to decide what kind of bridge to build. They also have standard bridge designs that they adapt for most routine bridge projects.

Even though software engineering is still a maturing discipline, many principles have become well established. We discuss these principles throughout the book.

As for technology, we base our designs on Java, a language with wide acceptance. Furthermore, in Chapter 3 we present a *framework* – a collection of classes that forms the basic structure upon which many different applications can be built. We demonstrate how this framework can be used to rapidly build several different applications.

Applying well-understood principles and reusing designs means that we are building on the experience and work of others, rather than 'reinventing the wheel'. The creative task of the engineer is to put knowledge to use in innovative ways to solve problems. This contrasts with the role of the scientist, which is to seek out new knowledge.

Theme 3: object orientation

Object-oriented (*OO*) techniques are based on the use of classes that act as abstractions of data, and that contain a set of procedures which act on that data. It is now widely recognized that object orientation is an effective design approach to manage the complexity inherent in most large systems.

In this book we discuss three major areas of software engineering in an object-oriented context: analysis, design and programming. In Chapter 2, we review basic OO principles and OO programming; then, in the rest of the book, we approach analysis and design from a primarily OO perspective. We will ask you to implement your designs in the OO language Java, so that you can see the consequences of your design decisions.

Theme 4: visual modeling using UML

The Unified Modeling Language (UML) is a set of notations for representing software requirements and design. It is now widely accepted as the standard approach to representing many aspects of software.

We will teach you in some detail how to use several different aspects of UML, including class diagrams (Chapter 5), state diagrams and interaction diagrams (both in Chapter 8).

Theme 5: evaluation of alternatives in requirements and design

There is rarely a single straightforward answer to any problem in software engineering. Whether you are developing requirements or performing design,

there are often several alternatives that must be assessed systematically to decide which is best.

In both requirements analysis and design we will encourage you to list alternatives, and discuss their advantages and disadvantages before making a decision. We will also encourage you to document your reasoning, frequently called *rationale*, so that others can understand your decisions.

Theme 6: incorporating quantitative and logical thinking

It is becoming increasingly necessary to incorporate mathematical thinking into software development. We will present basic ways to measure aspects of software systems and software engineering processes. The objective of doing this measurement is to help make predictions of development time and quality in order to better control these factors. This topic, commonly known as *software metrics*, is covered in the chapters on object-orientation (Chapter 2), requirements (Chapter 4), design (Chapter 9), testing (Chapter 10) and project management (Chapter 11).

We will also show several ways to make use of *logic* in order to develop software: in Chapter 5 we will introduce *OCL*, a language for formally describing properties of designs; and in Chapter 9 we will show how logic can be used in a technique called defensive programming.

Theme 7: iterative and agile development

Traditionally, software engineering has been performed following what is called the *waterfall* model. In this approach you first develop requirements; once these are complete you move on to design, and then to programming, testing and deployment. An outdated view held that you should completely finish each of these steps before moving on to the next; then, when you complete deployment, you are finished. In contrast, the currently accepted view is that software engineering is, and should be, a highly *iterative* process. So-called *agile* techniques are the most highly iterative of all (see the sidebar 'Agile versus conventional development' earlier in this chapter).

It is typical to develop the first iteration of a system as a *prototype*, with only rough requirements and little functionality. Doing this serves to help establish the requirements for the next iteration. Several iterations of prototypes may be needed before the stakeholders are finally satisfied with the requirements, at which time you can proceed with a more rigorous process involving more complete specification and design.

Even after delivering software to customers, you typically continue to build a series of new releases, each one involving most of the activities discussed in Section 1.6. Iterative development results in delivering smaller units of work (prototypes or releases) quite frequently. This means that the first release can be in the customers' hands earlier than if you had tried to develop a fully fledged system. It also means that if the system turns out to be a disaster, less work has been wasted.

We will practice the iterative approach in this book, starting in Chapter 3, by asking you to make a series of small changes to a project. You will do the requirements, design and implementation of each change, with changes becoming more sophisticated as you learn more of the material in the book.

We discuss processes the waterfall, iterative and other approaches in more detail in Chapter 11.

Theme 8: communicating effectively using documentation

Software engineers communicate with each other orally both in meetings and at each other's desks; however, it would never be possible to run a large project if all information had to be conveyed in this manner.

Agile documentation

Agile developers prefer to write very little documentation. Some would prefer that anything that needs documenting be put in code comments and nowhere else.

Writing clear documentation is therefore an essential skill. Documentation should be written at all stages of development and includes requirements, designs, user manuals, instructions for testers and project plans. One of the keys to writing good documentation is to understand the audience. You must provide the information the readers will need, and organize it in such a way that the readers can find it easily. For example, the audience for design documentation includes other software engineers with whom you are currently working, as well as those who will need to make changes later. Both groups need to understand what you did and why you did it.

Unfortunately, unless it is managed appropriately, writing documentation can waste resources and can be a source of rigidity in software development. The waste of resources can occur if documentation is never read – this will be the case if it is excessively voluminous, poorly written or not made readily available. Excessive documentation means that the readers cannot find what they want easily, and 'can't find the forest for the trees'. It is therefore as bad as if you had not created enough documentation to start with.

Forcing software developers to write documents prematurely just to meet specific deadlines can mean that the overall objective becomes writing documents, instead of solving problems. Furthermore, such documents can entrench poorly made decisions that are hard to change.

In this book, we will encourage you to write documentation but we will emphasize that it should be as short and succinct as possible, and it should serve the purpose of documenting your decisions and communicating them to others. Furthermore, documentation should be written in the context of risk management, discussed below, which means that it is always subject to change.

We will give you outlines of each type of document as well as several example documents. You will have the opportunity to practice writing the documents and also reviewing them in groups.

When writing documentation you should also be aware that there are often standards that you should adhere to. It is important that documentation used within a company have a standard format so that people can more easily use it.

Theme 9: risk management in all software engineering activities

Whereas documentation allows future readers to keep an eye on the past, we must also constantly keep an eye on the future. *Risk analysis* is a key software engineering activity in which we constantly assess any new information to determine whether it will cause problems for the project. If you believe there is a significant risk that a certain type of problem will arise, then you can take steps to reduce the risk.

Software is an investment that should provide benefits; and risks are natural in any investment. The objective must be to reduce risks to acceptable levels, while still achieving the benefits. Taking action to reduce risks is like adjusting your investment portfolio. Sometimes you put more effort into certain tasks to ensure the project is completed successfully; at other times you must cut parts of the system to avoid losses.

The last numbered section of every chapter will discuss the difficulties and risks to be considered in the material covered by that chapter. In the next section we begin this process by reviewing the most important risks in software engineering.

1.9 Difficulties and risks in software engineering as a whole

The following is a selection of general factors, or challenges, that can have a major impact on the success of a software engineering project. Software engineers should regularly analyze whether any of these poses a risk, and take the suggested corrective action if necessary.

Some of these points serve as a review of what we said earlier in this chapter. We will discuss many of them in more detail in subsequent chapters.

After each challenge listed below, we list some suggestions for resolution. These suggestions can be used both to reduce risk and solve problems. However, since each situation is different, the suggestions will not always work – experience and good judgment must be your ultimate guide. As you read through the rest of the book, you will learn more details about how to go about resolving the difficulties.

- **Complexity and large numbers of details**. Software systems tend to become complex because: a) it is easy to add new features, b) software developers typically add features without fully understanding a system, and c) the system may not have been originally designed to accommodate the features.
  Resolution. Design the system for flexibility right from the start. Divide the system into smaller subsystems, so that each one is naturally simpler. Resist the urge to add new features, and consider removing those that are not needed. Use tools designed to help you more fully understand the structure of a software system. Budget sufficient time to learn about the software before making changes. When faced with an over-complex system, redesign parts of it as necessary.

- **Uncertainty about technology**. You can never be sure whether the technology on which a system depends will work as expected. Hardware tends to be reliable, but special-purpose hardware or future versions of the hardware may differ from what you expect. Software libraries and other software systems with which a system interacts can be expected to have bugs and incompatibilities.
  Resolution. Avoid technology sold by just a single vendor and which has relatively few other customers. Widely used technology is more likely to be supported and to have had its defects removed. Avoid obscure features of any technology. Balance the benefits of your use of third-party technology with the risks of problems. Create prototypes to try out the technology you will be using.

- **Uncertainty about requirements**. Until a system is delivered and in use, you can never be quite sure whether it meets the customer's needs.
  Resolution. Understand the application domain so you can communicate effectively with clients and users. Follow a good requirements gathering and analysis process. Prototype to get an early view of potential problems. Continually interact with users and clients to keep up to date on their needs. Design with change in mind.

- **Uncertainty about software engineering skills**. Software engineering is heavily labor-intensive; however, skills of team members can vary dramatically and probably are the biggest single factor affecting success of a project.
  Resolution. Make sure software engineers have sufficient general education, plus training in the technology to be used. Make sure they have sufficient experience by 'practicing' on prototypes or systems that are of lesser importance. Put in place a mentoring system so that the software engineers can effectively learn from others.

- **Constant change**. Both technology and requirements can be expected to change regularly.
  Resolution. Design for flexibility to accommodate potential changes. Stay aware of things that may change. Adjust the requirements or design as soon as important changes are discovered. Avoid changing too much too frequently, however.

- **Deterioration of software design**. Software deteriorates due to successive changes that introduce bugs.
  Resolution. Build flexibility and other aspects of maintainability into the software from the start so that changes are easier to make. Ensure software engineers have sufficient training. Ensure changes are not rushed. Perform quality assurance activities on each change.

- **'Political' risks**. Not everybody will be happy with the requirements. Not everybody may want the system. Competition or organizational changes might render the system less important or might result in project cancelation. Various stakeholders may not understand certain software-engineering practices and may want you to do things with which you disagree.

Resolution. Participate in promoting and marketing the project. Enhance your negotiating and other 'people' skills. Regularly evaluate how the system will impact all the stakeholders, and work closely with them to foster increased understanding of issues.

1.10 Summary

We have emphasized in this chapter that software engineering is an emerging engineering specialty in which you focus on solving a customer's problem by developing high-quality software.

Since software is relatively intangible, our ability to work with it is different from other engineering products. It is possible for a beginner to rapidly program a significantly sized system, make changes to source code in a matter of minutes, and distribute thousands of copies at little cost. Unfortunately, developing systems in a rapid and ad hoc way like this leads to excessive complexity and increasing numbers of problems.

To perform good software engineering, it is necessary to incorporate discipline into software development. Some ways of doing this include carefully understanding users and their requirements, taking time to perform design, and carefully evaluating the quality of the software. You also must keep systems small at first to reduce the risk of failure, focus on delivering systems within a fixed amount of time, and constantly reassess what you are doing so that you can take action when problems arise.

Throughout the rest of this book we will present many different software engineering techniques so that you can learn how to achieve the goal of solving customers' problems more effectively.

1.11 For more information

At the end of each chapter we will discuss sources of information that you can consult to learn more about the material in that chapter. In this chapter, we list general software engineering resources; in later chapters we list resources covering specific issues.

The resources include web sites, books and periodicals. We have only listed web sites that we believe to contain reasonably reliable information or useful sets of links, which have stood the test of time, and are likely to be maintained. This book's web site (www.lloseng.com) contains a page with all the links shown in the book, updated as necessary.

Software engineering magazines published by major organizations

- *IEEE Software*, http://www.computer.org/software/
  The IEEE Computer Society is one of the two most important international organizations that focus on software engineering. They produce many software

engineering publications, but *IEEE Software* is probably the one most readable by practitioners.

- *IEEE Computer*, http://www.computer.org/computer/
  Also published by the IEEE Computer Society, this magazine covers a broader spectrum of computing topics, including software engineering. All members of the society receive this.

- *Communications of the ACM*, http://www.acm.org
  The Association for Computing Machinery (ACM) is the other main international organization involved in the development of software engineering. *CACM* is not exclusively about software engineering, but has many articles on this topic. It is included with membership in the ACM.

Other selected software engineering Internet sites

- The Software Engineering Body of Knowledge (SWEBOK), www.swebok.org
  The goal of this project, initiated by the ACM and the IEEE, is to gather together all the most important and widely accepted knowledge in software engineering. The SWEBOK initiative is under continuous development, and is an excellent resource to find detailed background material about the field.

- The ACM/IEEE software engineering code of ethics, http://www.acm.org/serving/se/code.htm

- The Community for Software Engineers, www.software-engineer.org

- The Wikipedia entry for software engineering: http://en.wikipedia.org/wiki/Software_engineering

- The Software Engineering Institute (SEI) at Carnegie Mellon University, www.sei.cmu.edu
  One of the foremost research institutes on software engineering.

General software engineering books

- Roger Pressman, *Software Engineering: a Practitioner's Approach*, 6th edition, McGraw Hill, 2004. This is one of the classic books covering all areas of software engineering in considerable depth. http://www.rspa.com/about/sepa.html

- Stephen R. Schach, *Object-Oriented and Classical Software Engineering*, 6th edition, McGraw Hill, 2004. http://www.mhhe.com/catalogs/0072865512.mhtml

- Ian Sommerville, *Software Engineering*, 7th Edition, Addison-Wesley, 2004, http://www.software-engin.com/

- Bernd Bruegge and Allen Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, 2nd edition, Prentice Hall, 2004

- Shari Lawrence Pfleeger, *Software Engineering: Theory and Practice*, 2nd edition, Prentice Hall, 2001.

The profession of engineering

- Greatest achievements of engineering: http://www.greatachievements.org
- Professional Engineering Institutions (UK): http://www.pei.org.uk
- Canadian Council of Professional Engineers: http://www.ccpe.ca
- National Society of Professional Engineers (US): http://www.nspe.org