CHAPTER

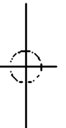
1 Software – problems and prospects

This chapter:

- reviews the goals of software engineering
- describes the difficulties of constructing large-scale software
- analyses the problems that software engineers face.

1.1 • Introduction

Software Engineering is about methods, tools and techniques used for developing software. This particular chapter is concerned with the reasons for having a field of study called software engineering, and with the problems that are encountered in developing software. This book as a whole explains a variety of techniques that attempt to solve the problems and meet the goals of software engineering.



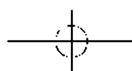
Software surrounds us everywhere in the industrialized nations – in domestic appliances, communications systems, transportation systems and in businesses. Software comes in different shapes and sizes – from the program in a mobile phone to the software to design a new automobile. In categorizing software, we can distinguish two major types:

- *system software* is the software that acts as tools to help construct or support applications software. Examples are operating systems, databases, networking software, compilers.
- *applications software* is software that helps perform some directly useful or enjoyable task. Examples are games, the software for automatic teller machines (ATMs), the control software in an airplane, e-mail software, word processors, spreadsheets.

Within the category of applications software, it can be useful to identify the following categories of software:

- games
- information systems – systems that store and access large amounts of data, for example, an airline seat reservation system

www.it-ebooks.info



- real-time systems – in which the computer must respond quickly, for example, the control software for a power station
- embedded systems – in which the computer plays a smallish role within a larger system, for example, the software in a telephone exchange or a mobile phone. Embedded systems are usually also real-time systems
- office software – word processors, spreadsheets, e-mail
- scientific software – carrying out calculations, modeling, prediction, for example, weather forecasting.

Software can either be off-the-shelf (e.g. Microsoft Word) or tailor-made for a particular application (e.g. software for the Apollo moon shots). The latter is sometimes called bespoke software.

All these types of software – except perhaps information systems – fall within the remit of software engineering. Information systems have a different history and, generally, different techniques are used for their development. Often the nature of the data (information) is used to dictate the structure of the software, so that analysis of the data is a prime step, leading to the design of the database for the application. This approach to software development is outside the scope of this book.

Constructing software is a challenging task, essentially because software is complex. The perceived problems in software development and the goals that software development seeks to achieve are:

- meeting users' needs
- low cost of production
- high performance
- portability
- low cost of maintenance
- high reliability
- delivery on time.

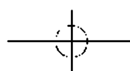
Each goal is also considered to be a problem because software engineering has generally been rather unsuccessful at reaching them. We will now look at each of these goals in turn. Later we will look at how the goals relate one to another.

In the remainder of this book we shall see that the development of particular types of software requires the use of special techniques, but many development techniques have general applicability.

1.2 ● Meeting users' needs

It seems an obvious remark to make that a piece of software must do what its users want of it. Thus, logically, the first step in developing some software is to find out what the client, customer or user needs. This step is often called requirements analysis or requirements engineering. It also seems obvious that it should be carried out with some care.

www.it-ebooks.info



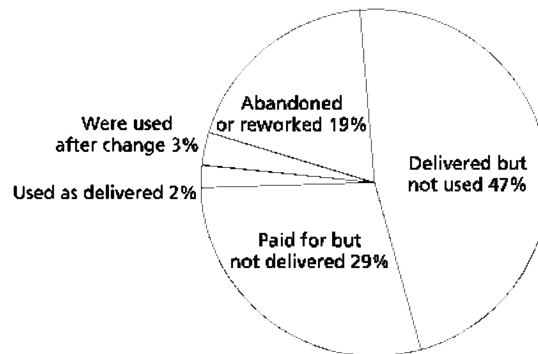


Figure 1.1 Effectiveness of typical large software projects

There is evidence, however, that this is not always the case. As evidence, one study of the effectiveness of large-scale software projects, Figure 1.1, found that less than 2% were used as delivered.

These figures are one of the few pieces of hard evidence available, because (not surprisingly) organizations are rather secretive about this issue. Whatever the exact figures, it seems that a large proportion of systems do not meet the needs of their users and are therefore not used as supplied. It may be, of course, that smaller systems are more successful.

We might go further and deduce that the *main* problem of software development lies in requirements analysis rather than in any other areas, such as reliability or cost, which are discussed below.

The task of trying to ensure that software does what its users want is known as ***validation***.

1.3 • The cost of software production

Examples of costs

First of all, let us get some idea of the scale of software costs in the world. In the USA it is estimated that about $500 billion are spent each year on producing software. This amounts to 1% of the gross national product. The estimated figure for the world is that $1,000 billion is spent each year on software production. These figures are set to rise by about 15% each year. The operating system that IBM developed for one of its major range of computers (called OS 360) is estimated to have cost $200 million. In the USA, the software costs of the manned space program were $1 billion between 1960 and 1970.

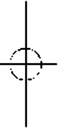
These examples indicate that the amount spent on software in the industrialized nations is of significant proportions.

www.it-ebooks.info

Programmer productivity

The cost of software is determined largely by the productivity of the programmers and the salaries that they are paid. Perhaps surprisingly, the productivity of the average programmer is only about 10–20 programming language statements per day. To the layperson, a productivity of 20 lines of code per day may well seem disgraceful. However, this is an average figure that should be qualified in two ways. First, enormous differences between individual programmers – factors of 20 – have been found in studies. Second, the software type makes a difference: applications software can be written more quickly than systems software. Also, this apparently poor performance does not just reflect the time taken to carry out coding, but also includes the time required to carry out clarifying the problem specification, software design, coding, testing and documentation. Therefore, the software engineer is involved in many more tasks than just coding. However, what is interesting is that the above figure for productivity is independent of the programming language used – it is similar whether a low-level language is used or a high-level language is used. It is therefore more difficult than it initially appears to attribute the level of productivity to laziness, poor tools or inadequate methods.

1.1 A well-known word processor consists of a million lines of code. Calculate how many programmers would be needed to write it, assuming that it has to be completed in five years. Assuming that they are each paid $50,000 per year, what is the cost of development?



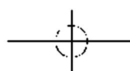
Predicting software costs

It is very difficult to predict in advance how long it will take to write a particular piece of software. It is not uncommon to underestimate the required effort by 50%, and hence the cost and delivery date of software is also affected.

Hardware versus software costs

The relative costs of hardware and software can be a lively battleground for controversy. In the early days of computers, hardware was costly and software relatively cheap. Nowadays, thanks to mass production and miniaturization, hardware is cheap and software (labor intensive) is expensive. So the costs of hardware and software have been reversed. These changes are reflected in the so-called "S-shaped curve", Figure 1.2, showing the relative costs as they have changed over the years. Whereas, in about 1955, software cost typically only about 10% of a project, it has now escalated to 90%, with the hardware comprising only 10%. These proportions should be treated carefully. They hold for certain projects only and not in each and every case. In fact, figures of this kind are derived largely from one-off large-scale projects.

www.it-ebooks.info



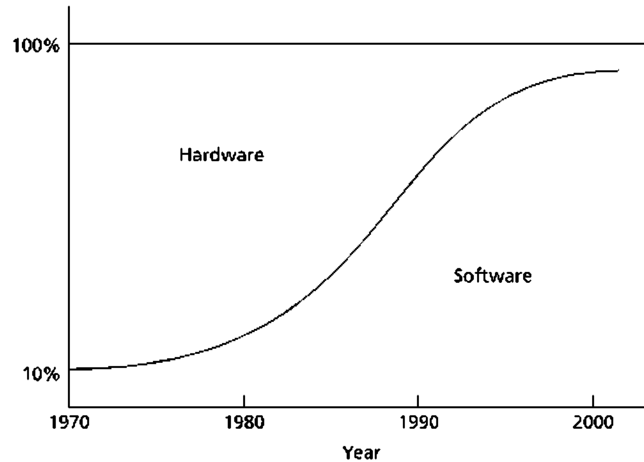


Figure 1.2 Changes in the relative costs of hardware and software

1.2 Someone buys a PC, with processor, monitor, hard disk and printer. They also buy an operating system and a word processing package. Calculate the relative costs of hardware and software.

We will now look at a number of issues that affect the popular perception of software and its costs.

The impact of personal computers

Perhaps the greatest influence on popular perceptions of software costs has come about with the advent of personal computing. Many people buy a PC for their home, and so come to realize very clearly what the costs are.

First is the "rock and roll" factor. If you buy a stereo for $200, you don't expect to pay $2,000 for a CD. Similarly, if you buy a PC for $1,000, you don't expect to pay $10,000 for the software, which is what it would cost if you hired a programmer to write it for you. So, of course, software for a PC either comes free or is priced at about $50 or so. It can be hard to comprehend that something for which you paid $50 has cost millions of dollars to develop.

Second is the teenager syndrome. Many school students write programs as part of their studies. So a parent might easily think, "My kid writes computer programs. What's so hard about programming? Why is software so expensive?"

www.it-ebooks.info

Software packages

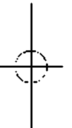
There has been another significant reaction to the availability of cheap computers. If you want to calculate your tax or design your garden, you can buy a program off the shelf to do it. Such software packages can cost as little as $50. The reason for the remarkably low price is, of course, that the producers of the software sell many identical copies – the mass production of software has arrived. The problem with an off-the-shelf package is, of course, that it may not do *exactly* what you want it to do and you may have to resort to tailor-made software, adapt your way of life to fit in with the software, or make do with the inadequacies.

Nonetheless, the availability of cheap packages conveys the impression that software is cheap to produce.

Application development tools

If you want to create certain types of applications software very quickly and easily, several development tools are available. Notable examples of these tools are Visual Basic and Microsoft Access. These tools enable certain types of program to be constructed very easily, and even people who are not programmers can learn to use tools like a spreadsheet (e.g. Microsoft Excel). Thus a perception is created that programming is easy and, indeed, that programmers may no longer be necessary.

The truth is, of course, that some software is very simple and easy to write, but most commercially used software is large and extremely complex.



The IT revolution

The sophistication of today's software far outstrips that of the past. For example, complex graphical user interfaces (GUI's) are now seen as essential, systems are commonly implemented on the web, and the sheer size of projects has mushroomed. People and organizations expect ever more facilities from computers. Arguably, as hardware becomes available to make previously impractical software projects feasible, software costs can only continue to escalate.

In summary, what we see today is that software is expensive:

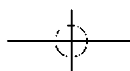
- relative to the gross national product
- because developers exhibit apparently low productivity
- relative to the cost of hardware
- in popular perception.

How is the cost made up?

It is interesting to see which parts of a software development project cost most money. Figure 1.3 shows typical figures.

Clearly the cost of testing is enormous, whereas coding constitutes only a small part of software development. One interpretation of this is that if a new magical development

www.it-ebooks.info



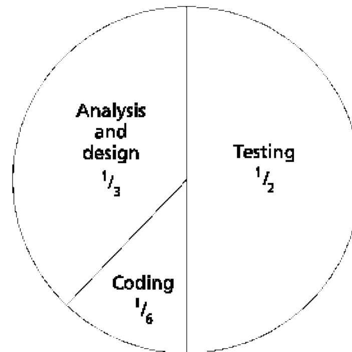


Figure 1.3 Relative costs of the stages of software development

method was devised that ensured the software was correct from the start, then testing would be unnecessary, and therefore only half the usual effort would be needed. Such a method would be a discovery indeed!

If mistakes are a major problem, when are they made? Figure 1.4 shows figures showing the number of errors made at the various stages of a typical project:

However, this data is rather misleading. What matters is how much it costs to *fix* a fault. And the longer the fault remains undiscovered, the more a fault costs to fix. Errors made during the earlier stages of a project tend to be more expensive, unless they are discovered almost immediately. Hence Figure 1.5 showing the relative costs of fixing mistakes in a typical project is probably more relevant.

A design flaw made early in the project may not be detected until late on in system testing – and it will certainly involve a whole lot of rework. By contrast, a syntax error in a program made late in the project will be automatically detected on a first compilation and then easily corrected.

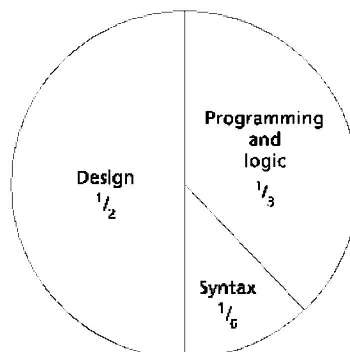


Figure 1.4 Relative numbers of errors made during the stages of software development

www.it-ebooks.info

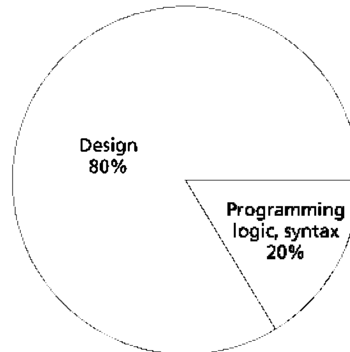


Figure 1.5 Relative cost of fixing different types of fault

1.4 ● Meeting deadlines

Meeting deadlines has always been a headache in software production. For example, surveys have consistently shown that this is the worst problem faced by project managers. The problem is related to the difficulty of predicting how long it will take to develop something. If you do not know how long it is going to take, you cannot hope to meet any deadline. It is a common experience for a software project to run late and over budget, disappointing the client and causing despair among the software team. Evidence suggests that around 60% of projects exceed their initial budgets and around 50% are completed late. Whatever the exact figures, meeting deadlines is clearly a problem.

Back in the 1980s, IBM's major new operating system (called OS 360) for its prime new range of computers was one year late. The computers themselves languished in warehouses waiting to be shipped. Microsoft's NT operating system was allegedly also a year late.

1.5 ● Software performance

This is sometimes called efficiency. This terminology dates from the days when the cost and speed of hardware meant that every effort was made to use the hardware primarily memory and processor as carefully as possible. More recently a cultural change has come about due to the increasing speed of computers and the fall in their cost. Nowadays there is more emphasis on meeting people's requirements, and consequently we will not spend much time on performance in this book. Despite this, performance cannot be completely ignored often we are concerned to make sure that:

- an interactive system responds within a reasonably short time
- a control signal is output to a plant in sufficient time
- a game runs sufficiently fast that the animation appears smooth
- a batch job is not taking 12 hours when it should take one.

www.it-ebooks.info

1.3 Identify two further software systems in which speed is an important factor.

The problem with fast run time and small memory usage is that they are usually mutually contradictory. As an example to help see how this comes about, consider a program to carry out a calculation of tax. We could either carry out a calculation, which would involve using relatively slow machine instructions, or we could use a lookup table, which would involve a relatively fast indexing instruction. The first case is slow but small, and the second case is fast but large. Generally, of course, it is necessary to make a judgment about what the particular performance requirements of a piece of software are.

1.6 • Portability

The dream of portability has always been to transfer software from one type of computer to another with the minimum expenditure of effort. With the advent of high-level languages and the establishment of international standards, the prospects looked bright for the complete portability of applications programs.

The reality is that market forces have dominated the situation. A supplier seeks to attract a customer by offering facilities over and above those provided by the standard language. Typically these may lessen the work of software development. An example is an exotic file access method. Once the user has bought the system, he or she is locked into using the special facilities and is reliant on the supplier for developments in equipment that are fully compatible. The contradiction is, of course, that each and every user is tied to a particular supplier in this way, and can only switch allegiance at a considerable cost in converting software. Only large users, like government agencies, are powerful enough to insist that suppliers adopt standards.

Given this picture of applications software, what are the prospects for systems software, like operating systems and filing systems, with their closer dependence on specific hardware?

1.7 • Maintenance

Maintenance is the term for any effort that is put into a piece of software after it has been written and put into operation. There are two main types:

- *remedial maintenance*, which is the time spent correcting faults in the software (fixing bugs)
- *adaptive maintenance*, which is modifying software either because the users' needs have changed or because, for example, the computer, operating system or programming language has changed

www.it-ebooks.info

Remedial maintenance is, of course, a consequence of inadequate testing. As we shall see, effective testing is notoriously difficult and time-consuming, and it is an accepted fact of life in software engineering that maintenance is inevitable.

It is often difficult to predict the future uses for a piece of software, and so adaptive maintenance is also rarely avoided. But because software is called soft, it is sometimes believed that it can be modified easily. In reality, software is brittle, like ice, and when you try to change it, it tends to break rather than bend.

In either case, maintenance is usually regarded as a nuisance, both by managers, who have to make sure that there are sufficient people to do it, and by programmers, who regard it as less interesting than writing new programs.

Some idea of the scale of what has been called the "maintenance burden" can be appreciated by looking at a chart, Figure 1.6, showing typical figures for the amount of time spent in the different activities of developing a particular piece of software.

In a project like this, the maintenance effort is clearly overwhelming. It is not unusual for organizations that use well-established computer systems to be spending three-quarters of their programming time on maintenance.

Here are some more estimates:

- world-wide, there are 50 billion lines of Cobol in use today
- in the United States, 2% of the GNP is spent on software maintenance
- in the UK, £1 billion (about $1.5 million) annually are spent on software maintenance

The millions of lines of program written in what many people consider to be outdated programming languages (like Cobol) constitute what are known as ***legacy systems***. These are software systems that are up to 30 years old, but in full use in organizations today. They are often poorly documented, either because there was no documentation

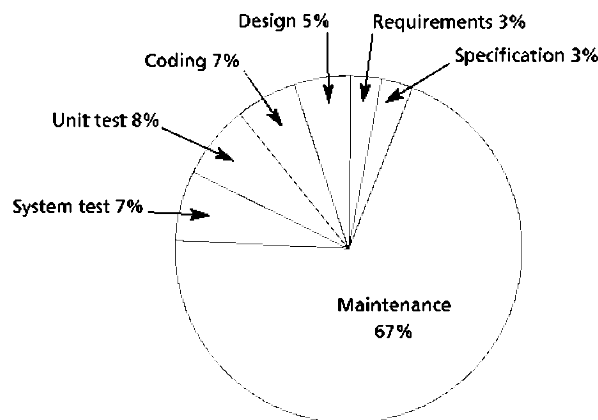
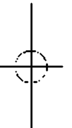


Figure 1.6 Relative costs of the stages of software development

www.it-ebooks.info

in the first place or because the documentation is useless because it has not been kept up to date as changes have been made. Legacy systems have been written using expertise in tools and methods that are rarely available today. For these reasons, it is expensive to update them to meet ever-changing requirements. Equally, it would be expensive to rewrite them from scratch using a contemporary language and methods. Thus legacy systems are a huge liability for organizations.

Another major example of the problems of maintenance was the ***millennium bug***. A great deal of software was written when memory was in short supply and expensive. Dates were therefore stored economically, using only the last two digits of the year, so that, for example, the year 1999 was stored as 99. After 2000, a computer could treat the date value 99 as 1999, 2099 or even 0099. The problem is that the meaning that is attached to a year differs from one system to another, depending on how the individual programmer decided to design the software. The only way to make sure that a program worked correctly after the year 2000 (termed year 2000 compliance) was to examine it line by line to find any reference to a date and then to fix the code appropriately. This was an immensely time-consuming, skilled and therefore costly process. The task often needed knowledge of an outdated programming language and certainly required an accurate understanding of the program's logic. The penalties for not updating software correctly are potentially immense, as modern organizations are totally reliant on computer systems for nearly all of their activities.



1.8 • Reliability

A piece of software is said to be reliable if it works, and continues to work, without crashing and without doing something undesirable. We say that software has a bug or a fault if it does not perform properly. We presume that the developer knew what was required and so the unexpected behavior is not intended. It is common to talk about bugs in software, but it is also useful to define some additional terms more clearly:

- *error* – a wrong decision made during software development
- *fault* – a problem that may cause software to depart from its intended behavior
- *failure* – an event when software departs from its intended behavior.

In this terminology, a fault is the same as a bug. An error is a mistake made by a human being during one of the stages of software development. An error causes one or more faults within the software, its specification or its documentation. In turn, a fault can cause one or more failures. Failures will occur while the system is being tested and after it has been put into use. (Confusingly, some authors use the terms fault and failure differently.) Failures are the symptom that users experience, whereas faults are a problem that the developer has to deal with. A fault may never manifest itself because the conditions that cause it to make the software fail never arise. Conversely a single fault may cause many different and frequent failures.

www.it-ebooks.info

The job of removing bugs and trying to ensure reliability is called *verification*.

There is a close but distinct relationship between the concept of reliability and that of meeting users' needs, mentioned above. Requirements analysis is concerned with establishing clearly what the user wants. *Validation* is a collection of techniques that try to ensure that the software does meet the requirements. On the other hand, reliability is to do with the technical issue of whether there are any faults in the software.

Currently testing is one of the main techniques for trying to ensure that software works correctly. In testing, the software is run and its behavior checked against what is expected to happen. However, as we shall see later in this book, there is a fundamental problem with testing: however much you test a piece of software, you can never be sure that you have found every last bug. This leads us to assert fairly confidently the unsettling conclusion that every large piece of software contains errors.

The recognition that we cannot produce bug-free software, however hard we try, has led to the concept of *good enough software*. This means that the developer assesses what level of faults are acceptable for the particular project and releases the software when this level is reached. By level, we mean the number and severity of faults. For some applications, such as a word processor, more faults are acceptable than in a safety critical system, such as a drive-by-wire car. Note that this means that good enough software is sold or put into productive use knowing that it contains bugs.

On the other hand, another school of thought says that if we can only be careful enough, we can create *zero defect software* – software that is completely fault free. This approach involves the use of stringent quality assurance techniques that we will examine later in this book.

One way of gauging the scale of the reliability problem is to look at the following series of cautionary tales.

In the early days of computing – the days of batch data-processing systems – it used to be part of the folklore that computers were regularly sending out fuel bills for (incorrectly) enormous amounts. Although the people who received these bills might have been seriously distressed, particularly the old, the situation was widely regarded as amusing. Reliability was not treated as an important issue.

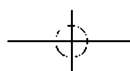
IBM's major operating system OS 360 had at least 1,000 bugs each and every time it was rereleased. How is this known (after all we would expect that IBM would have corrected all known errors)? The answer is that by the time the next release was issued, 1,000 errors had been found in the previous version.

As part of the US space program, an unmanned vehicle was sent to look at the planet Venus. Part way through its journey a course correction proved necessary. A computer back at mission control executed the following statement, written in the Fortran language:

```
DO 3 I = 1.3
```

This is a perfectly valid Fortran statement. The programmer had intended it to be a repetition statement, which is introduced by the word **DO**. However, a **DO** statement should

www.it-ebooks.info



contain a comma rather than the period character actually used. The use of the period makes the statement into assignment statement, placing a value 1.3 into the variable named DO3I. The space probe turned on the wrong course and was never seen again. Thus small errors can have massive consequences. Note that this program had been compiled successfully without errors, which illustrates how language notation can be important. Bugs can be syntactically correct but incorrect for the job they are required for. The program had also been thoroughly tested, which demonstrates the limitations of testing techniques.

In March 1979, an error was found in the program that had been used to design the cooling systems of nuclear reactors in the USA. Five plants were shut down because their safety became questionable.

Some years ago, the USA's system for warning against enemy missiles reported that the USA was being attacked. It turned out to be a false alarm – the result of a computer error – but before the error was discovered, people went a long way into the procedures for retaliating using nuclear weapons. This happened not just once, but three times in a period of a year.

Perhaps the most expensive consequence of a software fault was the crash, 40 seconds after blast-off, of the European Space Agency's Ariane 5 launcher in June 1996. The loss was estimated at $500 million, luckily without loss of life.

In 1999, the website for eBay, the internet auction site went down for 22 hours. As the markets began to doubt that eBay could adequately maintain its key technology, $6 billion was wiped off the share value of the company.

The incidents related above are just a few in a long catalog of expensive problems caused by software errors over the years, and there is no indication that the situation is improving.

How does the reliability of software compare with the reliability of hardware? Studies show that where both the hardware and software are at comparable levels of development, hardware fails three times as often as software. Although this is grounds for friendly rivalry between software and hardware designers, it can be no grounds for complacency among software people.

There are particular applications of computers that demand particularly high reliability. These are known as *safety-critical systems*. Examples are:

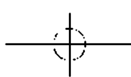
- fly-by-wire control of an aircraft
- control of critical processes, such as a power station
- control of medical equipment

In this book, we will look at techniques that can be used in developing systems such as these.

It is not always clear whether a piece of software is safety related. The example mentioned earlier of the faulty software used in designing a power plant is just one example. Another example is communications software that might play a critical role in summoning help in an emergency.

The conclusion is that, generally, software has a poor reputation for reliability.

www.it-ebooks.info



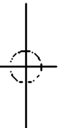
1.4 Identify three further examples of software systems that are safety critical and three that are not.

1.9 • Human–computer interaction

The user interface is what the human user of a software package sees when they need to use the software. There are many examples of computer systems that are not easy to use:

- many people have difficulty programming a video cassette recorder (VCR)
- some people find it difficult to divert a telephone call to another user within an organization

In recent years, many interfaces have become graphical user interfaces (GUIs) that use windows with features like buttons and scroll bars, together with pointing devices like a mouse and cursor. Many people saw this as a massive step in improving the user interface, but it remains a challenging problem to design a user interface that is simple and easy to use.



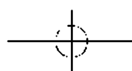
1.5 Think of two computer-based systems that you know of that are difficult to use in some way or another. Alternatively, think of two features of a program you use that are difficult to use.

1.10 • A software crisis?

We have discussed various perceived problems with software:

- it fails to do what users want it to do
- it is expensive
- it isn't always fast enough
- it cannot be transferred to another machine easily
- it is expensive to maintain
- it is unreliable
- it is often late
- it is not always easy to use.

www.it-ebooks.info



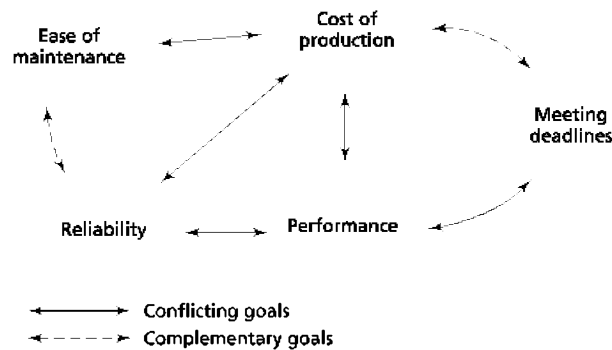


Figure 1.7 Complementary and conflicting goals in a software project

Of these, meeting users' needs (validation), reducing software costs, improving reliability (verification) and delivery on time are probably the four most important present-day problems.

Many people argue that things have been so bad – and continue to be so bad – that there is a continuing real "crisis" in software development. They argue that something must be done about the situation, and the main remedy must be to bring more scientific principles to bear on the problem – hence the introduction of the term software engineering. Indeed, the very term software engineering conveys that there is a weightier problem than arises in small-scale programming.

One of the obstacles to trying to solve the problems of software is that very often they conflict with each other. For example, low cost of construction and high reliability conflict. Again, high performance and portability are in conflict. Figure 1.7 indicates the situation.

Happily, some goals do not conflict with each other. For example, low cost of maintenance and high reliability are complementary.

1.11 • A remedy – software engineering?

As we have seen, it is generally recognized that there are big problems with developing software successfully. A number of ideas have been suggested for improving the situation. These methods and tools are collectively known as software engineering. Some of the main ideas are:

- greater emphasis on carrying out all stages of development systematically.
- computer assistance for software development – software tools.
- an emphasis on finding out exactly what the users of a system really want (requirements engineering and validation)
- demonstrating an early version of a system to its customers (prototyping)
- use of new, innovative programming languages

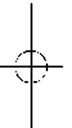
www.it-ebooks.info

- greater emphasis on trying to ensure that software is free of errors (verification).
- incremental development, where a project proceeds in small, manageable steps

We will be looking at all of these ideas in this book. These solutions are not mutually exclusive; indeed they often complement each other.

Verification, prototyping and other such techniques actually address only some of the problems encountered in software development. A large-scale software project will comprise a number of separate related activities, analysis, specification, design, implementation, and so on. It may be carried out by a large number of people working to strict deadlines, and the end product usually has to conform to prescribed standards. Clearly, if software projects are to have any chance of successfully delivering correct software on time within budget, they must be thoroughly planned in advance and effectively managed as they are executed. Thus the aim is to replace ad hoc methods with an organized discipline.

One term that is used a lot these days in connection with software is the word *quality*. One might argue that any product (from a cream bun to a washing machine) that fulfills the purpose for which it was produced could be considered to be a quality product. In the context of software, if a package meets, and continues to meet, a customer's expectations, then it too can be considered to be a quality product. In this perspective, quality can be attained only if effective standards, techniques and procedures exist to be applied, and are seen to be properly employed and monitored. Thus, not only do good methods have to be applied, but they also have to be seen to be applied. Such procedures are central to the activity called "quality assurance".



The problem of producing "correct" software can be addressed by using appropriate specification and verification techniques (formal or informal). However, correctness is just one aspect of quality; the explicit use of project management discipline is a key factor in achieving high-quality software.

Summary

We have considered a number of goals and problem areas in software development. Generally, software developers have a bad image, a reputation for producing software that is:

- late
- over budget
- unreliable
- inflexible
- hard to use.

Because the problems are so great, there has been widespread talk of a crisis in software production. The general response to these problems has been the creation of a number of systematic approaches, novel techniques and notations to address the software development task. The different methods, tools and languages fit within a plan of action (called a process model). This book is about these approaches. Now read on.

www.it-ebooks.info

This document was created with Win2PDF available at http://www.win2pdf.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.