1 Introduction

Objectives

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

- understand what software engineering is and why it is important;
- know the answers to key questions that provide an introduction to software engineering;
- understand some ethical and professional issues that are important for software engineers.

Contents

Virtually all countries now depend on complex computer-based systems. National infrastructures and utilities rely on computer-based systems and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerised, as is the financial system. Therefore, producing and maintaining software cost-effectively is essential for the functioning of national and international economies.

Software engineering is an engineering discipline whose focus is the cost-effective development of high-quality software systems. Software is abstract and intangible. It is not constrained by materials, or governed by physical laws or by manufacturing processes. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. However, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand.

The notion of *software engineering* was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis'. This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits. Their power made hitherto unrealisable computer applications a feasible proposition. The resulting software was orders of magnitude larger and more complex than previous software systems.

Early experience in building these systems showed that informal software development was not good enough. Major projects were sometimes years late. The software cost much more than predicted, was unreliable, was difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling whilst software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

These techniques have become part of software engineering and are now widely used. However, as our ability to produce software has increased, so too has the complexity of the software systems that we need. New technologies resulting from the convergence of computers and communication systems and complex graphical user interfaces place new demands on software engineers. As many companies still do not apply software engineering techniques effectively, too many projects still produce software that is unreliable, delivered late and over budget.

I think that we have made tremendous progress since 1968 and that the development of software engineering has markedly improved our software. We have a much better understanding of the activities involved in software development. We have developed effective methods of software specification, design and implementation. New notations and tools reduce the effort required to produce large and complex systems.

We know now that there is no single 'ideal' approach to software engineering. The wide diversity of different types of systems and organisations that use these systems means that we need a diversity of approaches to software development. However, fundamental notions of process and system organisation underlie all of these techniques, and these are the essence of software engineering.

Software engineers can be rightly proud of their achievements. Without complex software we would not have explored space, would not have the Internet and modern telecommunications, and all forms of travel would be more dangerous and expensive. Software engineering has contributed a great deal, and I am convinced that, as the discipline matures, its contributions in the 21st century will be even greater.

1.1 FAQs about software engineering

This section is designed to answer some fundamental questions about software engineering and to give you some impression of my views of the discipline. The format that I have used here is the 'FAQ (Frequently Asked Questions) list'. This approach is commonly used in Internet newsgroups to provide newcomers with answers to frequently asked questions. I think that it is a very effective way to give a succinct introduction to the subject of software engineering.

Figure 1.1 summarises the answers to the questions in this section.

1.1.1 What is software?

Many people equate the term *software* with computer programs. However, I prefer a broader definition where software is not just the programs but also all associated documentation and configuration data that is needed to make these programs operate correctly. A software system usually consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system and web sites for users to download recent product information.

Software engineers are concerned with developing software products, i.e., software which can be sold to a customer. There are two fundamental types of software product:

1. *Generic products* These are stand-alone systems that are produced by a development organisation and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.
2. *Customised (or bespoke) products* These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What is software engineering?	Software engineering is an engineering discipline which is concerned with all aspects of software production.
What is the difference between software engineering and computer science?	Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development, including hardware, software and process engineering. Software engineering is part of this process.
What is a software process?	A set of activities whose goal is the development or evolution of software.
What is a software process model?	A simplified representation of a software process, presented from a specific perspective.
What are the costs of software engineering?	Roughly 60% of costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are software engineering methods?	Structured approaches to software development which include system models, notations, rules, design advice and process guidance.
What is CASE (Computer-Aided Software Engineering)?	Software systems which are intended to provide automated support for software process activities. CASE systems are often used for method support.
What are the attributes of good software?	The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.

Figure 1.1 Frequently asked questions about software engineering

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

However, the line between these types of products is becoming increasingly blurred. More and more software companies are starting with a generic system and customising it to the needs of a particular customer. Enterprise Resource Planning (ERP) systems, such as the SAP system, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

1.1.2 What is software engineering?

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

1. *Engineering discipline* Engineers make things work. They apply theories, methods and tools where these are appropriate, but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognise that they must work to organisational and financial constraints, so they look for solutions within these constraints.
2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

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

1.1.3 What's the difference between software engineering and computer science?

Essentially, computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers.

Ideally, all of software engineering should be underpinned by theories of computer science, but in reality this is not the case. Software engineers must often use *ad hoc* approaches to developing the software. Elegant theories of computer science cannot always be applied to real, complex problems that require a software solution.

1.1.4 What is the difference between software engineering and system engineering?

System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system

deployment as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.).

System engineering is an older discipline than software engineering. People have been specifying and assembling complex industrial systems such as aircraft and chemical plants for more than a hundred years. However, as the percentage of software in systems has increased, software engineering techniques such as use-case modelling and configuration management are being used in the systems engineering process. I discuss system engineering in Chapter 2.

1.1.5 What is a software process?

A software process is the set of activities and associated results that produce a software product. There are four fundamental process activities (covered later in the book) that are common to all software processes. These are:

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

Different types of systems need different development processes. For example, real-time software in an aircraft has to be completely specified before development begins whereas, in e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organised in different ways and described at different levels of detail for different types of software. However, use of an inappropriate software process may reduce the quality or the usefulness of the software product to be developed and/or increase the development costs.

Software processes are discussed in more detail in Chapter 4, and the important topic of software process improvement is covered in Chapter 28.

1.1.6 What is a software process model?

A software process model is a simplified description of a software process that presents one view of that process. Process models may include activities that are part of the software process, software products and the roles of people involved in soft-

ware engineering. Some examples of the types of software process model that may be produced are:

1. *A workflow model* This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.
2. *A dataflow or activity model* This represents the process as a set of activities, each of which carries out some data transformation. It shows how the input to the process, such as a specification, is transformed to an output, such as a design. The activities here may represent transformations carried out by people or by computers.
3. *A role/action model* This represents the roles of the people involved in the software process and the activities for which they are responsible.

Most software process models are based on one of three general models or paradigms of software development:

1. *The waterfall approach* This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed-off', and development goes on to the following stage.
2. *Iterative development* This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that satisfies the customer s needs. The system may then be delivered. Alternatively, it may be reimplemented using a more structured approach to produce a more robust and maintainable system.
3. *Component-based software engineering (CBSE)* This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch. I discuss CBSE in Chapter 19.

I return to these generic process models in Chapter 4 and Chapter 17.

1.1.7 What are the costs of software engineering?

There is no simple answer to this question as the distribution of costs across the different activities in the software process depends on the process used and the type of software that is being developed. For example, real-time software usually requires more extensive validation and testing than web-based systems. However,

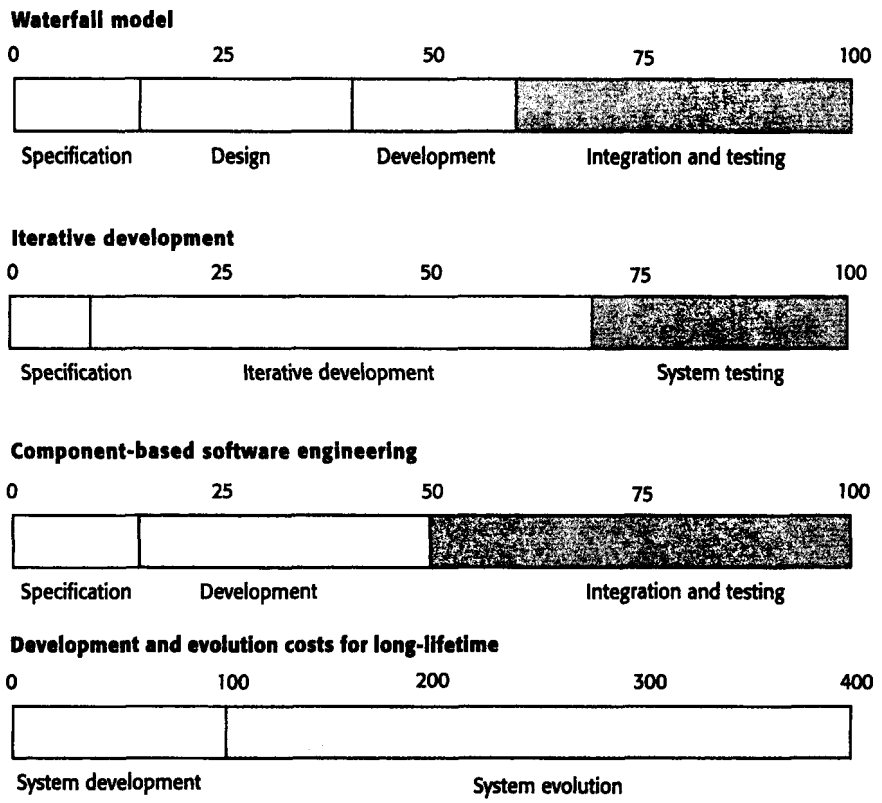


Figure 1.2 Software engineering activity cost distribution

each of the different generic approaches to software development has a different profile of cost distribution across the software process activities. If you assume that the total cost of developing a complex software system is 100 cost units then Figure 1.2 illustrates how these are spent on different process activities.

In the waterfall approach, the costs of specification, design, implementation and integration are measured separately. Notice that system integration and testing is the most expensive development activity. Normally, this is about 40% of the total development costs but for some critical systems it is likely to be at least 50% of the system development costs.

If the software is developed using an iterative approach, there is no hard line between specification, design and development. Specification costs are reduced because only a high-level specification is produced before development in this approach. Specification, design, implementation, integration and testing are carried out in parallel within a development activity. However, you still need an independent system testing activity once the initial implementation is complete.

Component-based software engineering has only been widely used for a short time. We don't have accurate figures for the costs of different software development activities in this approach. However, we know that development costs are reduced

relative to integration and testing costs. Integration and testing costs are increased because you have to ensure that the components that you use actually meet their specification and work as expected with other components.

On top of development costs, costs are also incurred in changing the software after it has gone into use. The costs of evolution vary dramatically depending on the type of system. For long-lifetime software systems, such as command and control systems that may be used for 10 years or more, these costs are likely to exceed the development costs by a factor of 3 or 4, as illustrated in the bottom bar in Figure 1.3. However, smaller business systems have a much shorter lifetime and correspondingly reduced evolution costs.

These cost distributions hold for customised software that is specified by a customer and developed by a contractor. For software products that are (mostly) sold for PCs, the cost profile is likely to be different. These products are usually developed from an outline specification using an evolutionary development approach. Specification costs are relatively low. However, because they are intended for use on a range of different configurations, they must be extensively tested. Figure 1.3 shows the type of cost profile that might be expected for these products.

The evolution costs for generic software products are particularly hard to estimate. In many cases, there is little formal evolution of a product. Once a version of the product has been released, work starts on the next release and, for marketing reasons, this is likely to be presented as a new (but compatible) product rather than as a modified version of a product that the user has already bought. Therefore, the evolution costs are not assessed separately as they are in customised software but are simply the development costs for the next version of the system.

1.1.8 What are software engineering methods?

A software engineering method is a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way. Methods such as Structured Analysis (DeMarco, 1978) and JSD (Jackson, 1983) were first developed in the 1970s. These methods attempted to identify the basic functional components of a system; function-oriented methods are still used. In the 1980s and 1990s, these function-oriented methods were supplemented by object-oriented (OO) methods such as those proposed by Booch (Booch, 1994) and Rumbaugh (Rumbaugh, et al., 1991). These different approaches have now been integrated into a single unified approach built around the Unified Modeling Language (UML) (Booch, et al., 1999; Rumbaugh, et al., 1999a; Rumbaugh, et al., 1999b).

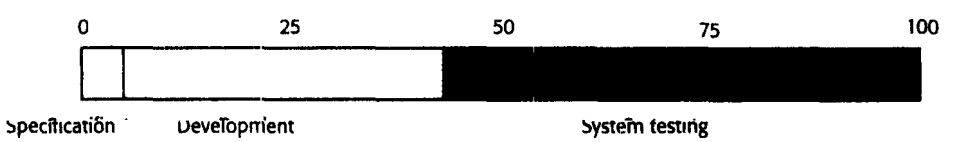


Figure 1.3 Product development costs

Component	Description	Example
System model descriptions	Descriptions of the system models which should be developed and the notation used to define these models.	Object models, data-flow models, state machine models, etc.
Rules	Constraints which always apply to system models.	Every entity in a system model must have a unique name.
Recommendations	Heuristics which characterise good design practice in this method. Following these recommendations should lead to a well-organised system model.	No object should have more than seven sub-objects associated with it.
Process guidance	Descriptions of the activities which may be followed to develop the system models and the organisation of these activities	Object attributes should be documented before defining the operations associated with an object.

Figure 1.4 Method components

There is no ideal method, and different methods have different areas where they are applicable. For example, object-oriented methods are often appropriate for interactive systems but not for systems with stringent real-time requirements.

All methods are based on the idea of developing models of a system that may be represented graphically and using these models as a system specification or design. Methods include a number of different components (Figure 1.4).

1.1.9 What is CASE?

The acronym CASE stands for Computer-Aided Software Engineering. It covers a wide range of different types of programs that are used to support software process activities such as requirements analysis, system modelling, debugging and testing. All methods now come with associated CASE technology such as editors for the notations used in the method, analysis modules which check the system model according to the method rules and report generators to help create system documentation. Th CASE tools may also include a code generator that automatically generates source code from the system model and some process guidance for software engineers.

1.1.10 What are the attributes of good software?

As well as the services that it provides, software products have a number of other associated attributes that reflect the quality of that software. These attributes are not directly concerned with what the software does. Rather, they reflect its behaviour while it is executing and the structure and organisation of the source program and associated documentation. Examples of these attributes (sometimes called non-functional attributes) are the software s response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure, an

Figure 1.5 Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.
Dependability	Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Usability	Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

interactive game must be responsive, a telephone switching system must be reliable, and so on. These can be generalised into the set of attributes shown in Figure 1.5, which, I believe, are the essential characteristics of a well-designed software system.

1.1.11 What are the key challenges facing software engineering?

Software engineering in the 21st century faces three key challenges:

1. *The heterogeneity challenge* Increasingly, systems are required to operate as distributed systems across networks that include different types of computers and with different kinds of support systems. It is often necessary to integrate new software with older legacy systems written in different programming languages. The heterogeneity challenge is the challenge of developing techniques for building dependable software that is flexible enough to cope with this heterogeneity.
2. *The delivery challenge* Many traditional software engineering techniques are time-consuming. The time they take is required to achieve software quality. However, businesses today must be responsive and change very rapidly. Their supporting software must change equally rapidly. The delivery challenge is the challenge of shortening delivery times for large and complex systems without compromising system quality.
3. *The trust challenge* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. The trust challenge is to develop techniques that demonstrate that software can be trusted by its users.

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

1.2 Professional and ethical responsibility

Like other engineering disciplines, software engineering is carried out within a legal and social framework that limits the freedom of engineers. Software engineers must accept that their job involves wider responsibilities than simply the application of technical skills. They must also behave in an ethical and morally responsible way if they are to be respected as professionals.

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

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

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

The ACM and the IEEE have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form, shown in Figure 1.6, and a longer form (Gotterbarn, et al., 1999) that adds detail and substance to the

Software Engineering Code of Ethics and Professional Practice

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

PREAMBLE

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

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

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

Figure 1.6 ACM/IEEE Code of Ethics (©IEEE/ACM 1999)

shorter version. The rationale behind this code is summarised in the first two paragraphs of the longer form:

> *Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.*
>
> *The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of*

the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer.

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

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

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

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

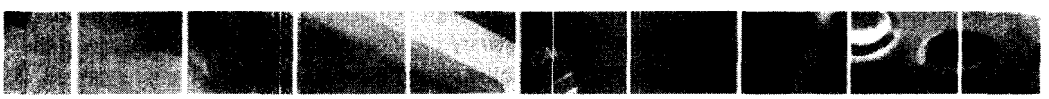
You must make up your own mind in these matters. The appropriate ethical position here depends entirely on the views of the individuals who are involved. In this case, the potential for damage, the extent of the damage and the people affected by the damage should influence the decision. If the situation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

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

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

The general area of ethics and professional responsibility is one that has received increasing attention over the past few years. It can be considered from a philosophical standpoint where the basic principles of ethics are considered, and software engineering ethics are discussed with reference to these basic principles. This is the approach taken by Laudon (Laudon, 1995) and to a lesser extent by Huff and Martin (Huff and Martin, 1995).

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



KEY POINTS

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software products consist of developed programs and associated documentation. Essential product attributes are maintainability, dependability, efficiency and acceptability.

  The software process includes all of the activities involved in software development. The high-level activities of software specification, development, validation and evolution are part of all software processes.

  Methods are organised ways of producing software. They include suggestions for the process to be followed, the notations to be used, system models to be developed and rules governing these models and design guidelines.
- CASE tools are software systems that are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests that have been run.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.

  Professional societies publish codes of conduct that set out the standards of behaviour expected of their members.

FURTHER READING

Fundamentals of Software Engineering. A general software engineering text that takes a rather different perspective on the subject than this book. (C. Ghezi, et. al., Prentice Hall, 2003.)

'Software engineering: The state of the practice'. A special issue of *IEEE Software* that includes several articles discussing current practice in software engineering, how this has changed and the extent to which new software technologies are used. (*IEEE Software*, 20 (6), November 2003.)

Software Engineering: An Engineering Approach. A general text that takes a rather different approach to my book but which includes a number of useful case studies. (J. F. Peters and W. Pedrycz, 2000, John Wiley & Sons.)

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

'Software engineering code of ethics is approved'. An article that discusses the background to the development of the ACM/IEEE Code of Ethics and includes both the short and long form of the code. (*Comm. ACM*, D. Gotterbarn, et al., October 1999.)

'No silver bullet: Essence and accidents of software engineering'. In spite of its age, this paper is a good general introduction to the problems of software engineering. The essential message of the paper, that there is no simple answer to the problems of software engineering, hasn't changed. (F. P. Brooks, *IEEE Computer*, 20 (4), April 1987.)

EXERCISES

1.1 By making reference to the distribution of software costs discussed in Section 1.1.6, explain why it is appropriate to consider software to be more than the programs that can be executed by end-users of a system.

1.2 What are the differences between generic software product development and custom software development?

1.3 What are the four important attributes which all software products should have? Suggest four other attributes that may sometimes be significant.

1.4 What is the difference between a software process model and a software process? Suggest two ways in which a software process model might be helpful in identifying possible process improvements.

1.5 Explain why system testing costs are particularly high for generic software products that are sold to a very wide market.

1.6 Software engineering methods became widely used only when CASE technology became

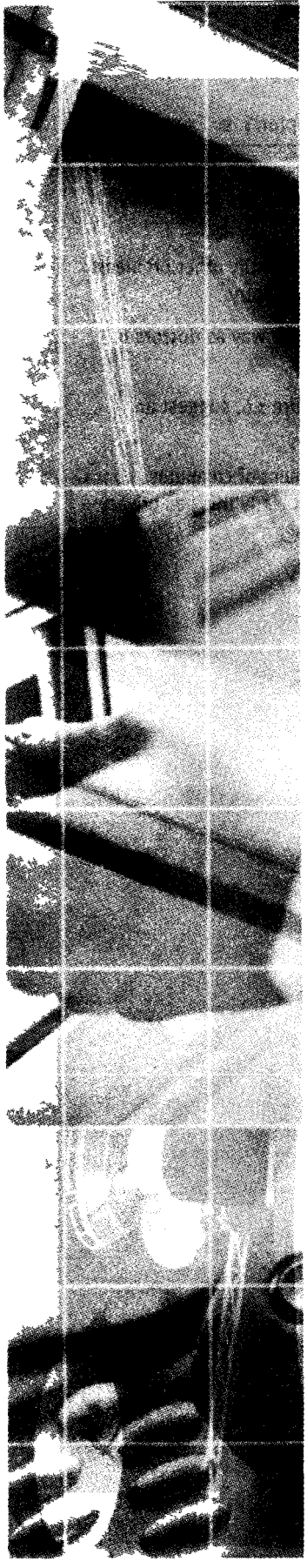
available to support them. Suggest five types of method support that can be provided by CASE tools.

1.7 Apart from the challenges of heterogeneity, rapid delivery and trust, identify other problems and challenges that software engineering is likely to face in the 21st century.

1.8 Discuss whether professional engineers should be certified in the same way as doctors or lawyers.

1.9 For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.6, suggest an appropriate example that illustrates that clause.

1.10 To help counter terrorism, many countries are planning the development of computer systems that track large numbers of their citizens and their actions. Clearly this has privacy implications. Discuss the ethics of developing this type of system.



2 Socio-technical systems

Objectives

The objectives of this chapter are to introduce the concept of a socio-technical system—a system that includes people, software and hardware—and to discuss the systems engineering process. When you have read this chapter, you will:

- know what is meant by a socio-technical system and understand the difference between a technical computer-based system and a socio-technical system;
- have been introduced to the concept of emergent system properties such as reliability, performance, safety and security;
- understand the activities that are involved in the systems engineering process;
- understand why the organisational context of a system affects its design and use;
- know what is meant by a 'legacy system', and why these systems are often critical to the operation of many businesses.

Contents

The term *system* is one that is universally used. We talk about computer systems, operating systems, payment systems, the educational system, the system of government, and so on. These are all obviously quite different uses of the word *system* although they share the characteristic that, somehow, the system is more than simply the sum of its parts.

Very abstract systems such as the system of government are well outside the scope of this book. Consequently, I focus here on systems that include computers and that have some specific purpose such as to enable communication, support navigation, and compute salaries. Therefore, a useful working definition of these types of systems is:

> *A system is a purposeful collection of interrelated components that work together to achieve some objective.*

This general definition embraces a vast range of systems. For example, a very simple system such as a pen may only include three or four hardware components. By contrast, an air traffic control system includes thousands of hardware and software components plus human users who make decisions based on information from the computer system.

Systems that include software fall into two categories:

- *Technical computer-based systems* are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones and most personal computer software. Individuals and organisations use technical systems for some purpose but knowledge of this purpose is not part of the system. For example, the word processor I am using is not aware that is it being used to write a book.
- *Socio-technical systems* include one or more technical systems but, crucially, also include knowledge of how the system should be used to achieve some broader objective. This means that these systems have defined operational processes, include people (the operators) as inherent parts of the system, are governed by organisational policies and rules and may be affected by external constraints such as national laws and regulatory policies. For example, this book was created through a socio-technical publishing system that includes various processes and technical systems.

Essential characteristics of socio-technical systems are as follows.

1. They have emergent properties that are properties of the system *as a whole* rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. As this is so complex, the emergent properties can only be evaluated once the system has been assembled.

2. They are often nondeterministic. This means that, when presented with a specific input, they may not always produce the same output. The system's behaviour depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behaviour.
3. The extent to which the system supports organisational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships and conflicts between organisational objectives and how people in the organisation interpret these objectives. New management may reinterpret the organisational objective that a system is designed to support, and a successful' system may then become a 'failure'.

In this book, I am concerned with socio-technical systems that include hardware and software, which have defined operational processes and which offer an interface, implemented in software, to human users. Software engineers should have some knowledge of socio-technical systems and systems engineering (White, et al., 1993; Thayer, 2002) because of the importance of software in these systems. For example, there were fewer than 10 megabytes of software in the US Apollo space program that put a man on the moon in 1969, but there are about 100 megabytes of software in the control systems of the Columbus space station.

A characteristic of all systems is that the properties and the behaviour of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of some other components. Thus, software can only operate if the processor is operational. The processor can only carry out computations if the software system defining these computations has been successfully installed.

Systems are usually hierarchical and so include other systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. These other systems are called *sub-systems*. A characteristic of sub-systems is that they can operate as independent systems in their own right. Therefore, the same geographical information system may be used in different systems.

Because software is inherently flexible, unexpected systems problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this ghosting. Their solution may be to enhance the image-processing capabilities of the software to remove the ghost images. This may slow down the software so that its performance becomes unacceptable. The problem may then be characterised as a 'software failure' whereas, in fact, it was a failure in the design process for the system as a whole.

This situation, where software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. Many so-called software failures were not a consequence of inherent software problems; they

were the result of trying to change the software to accommodate modified system engineering requirements. A good example of this was the failure of the Denver airport baggage system (Swartz, 1996), where the controlling software was expected to deal with several limitations in the equipment used.

Software engineering is therefore critical for the successful development of complex, computer-based socio-technical systems. As a software engineer, you should not simply be concerned with the software itself but you should also have a broader awareness of how that software interacts with other hardware and software systems and how it is supposed to be used. This knowledge helps you understand the limits of software, to design better software and to participate as equal members of a systems engineering group.

2.1 Emergent system properties

The complex relationships between the components in a system mean that the system is more than simply the sum of its parts. It has properties that are properties of the system as a whole. These *emergent properties* (Checkland, 1981) cannot be attributed to any specific part of the system. Rather, they emerge only once the system components have been integrated. Some of these properties can be derived directly from the comparable properties of sub-systems. However, more often, they result from complex sub-system interrelationships that cannot, in practice, be derived from the properties of the individual system components. Examples of some emergent properties are shown in Figure 2.1.

There are two types of emergent properties:

1. *Functional emergent properties* appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.
2. *Non-functional emergent properties* relate to the behaviour of the system in its operational environment. Examples of non-functional properties are reliability, performance, safety and security. These are often critical for computer-based systems, as failure to achieve some minimal defined level in these properties may make the system unusable. Some users may not need some system functions so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.

To illustrate the complexity of emergent properties, consider the property of system reliability. Reliability is a complex concept that must always be considered at the system level rather than at the individual component level. The components in

Figure 2.1 Examples of emergent properties

Property	Description
Volume	The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.
Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguard .
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty and modify or replace these components.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.

a system are interdependent, so failures in one component can be propagated through the system and affect the operation of other components. It is often difficult to anticipate how the consequences of component failures propagate through the system. Consequently, you cannot make good estimates of overall system reliability from data about the reliability of system components.

There are three related influences on the overall reliability of a system:

1. *Hardware reliability* What is the probability of a hardware component failing and how long does it take to repair that component?
2. *Software reliability* How likely is it that a software component will produce an incorrect output? Software failure is usually distinct from hardware failure in that software does not wear out. Failures are usually transient so the system carries on working after an incorrect result has been produced.
3. *Operator reliability* How likely is it that the operator of a system will make an error?

All of these are closely linked. Hardware failure can generate spurious signals that are outside the range of inputs expected by software. The software can then behave unpredictably. Operator error is most likely in conditions of stress, such as when system failures are occurring. These operator errors may further stress the hardware, causing more failures, and so on. Thus, the initial, recoverable failure can rapidly develop into a serious problem requiring a complete system shutdown.

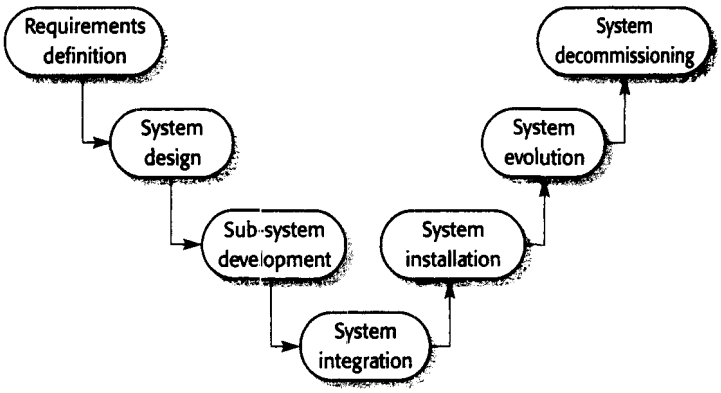


Figure 2.2 The systems engineering process

Like reliability, other emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, pose different problems. Here, you are not simply concerned with an attribute that is related to the overall behaviour of the system but are concerned with behaviour that the system should *not* exhibit. A secure system is one that does not allow unauthorised access to its data but it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these properties by default. That is, you only know that a system is insecure when someone breaks into it.

2.2 Systems engineering

Systems engineering is the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems. Systems engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment. They must think about the services that the system provides, the constraints under which the system must be built and operated and the ways in which the system is used to fulfil its purpose. As I have discussed, software engineers need an understanding of system engineering because problems of software engineering are often a result of system engineering decisions (Thayer, 1997; Thayer, 2002).

The phases of the systems engineering process are shown in Figure 2.2. This process was an important influence on the 'waterfall' model of the software process that I describe in Chapter 4.

There are important distinctions between the system engineering process and the software development process:

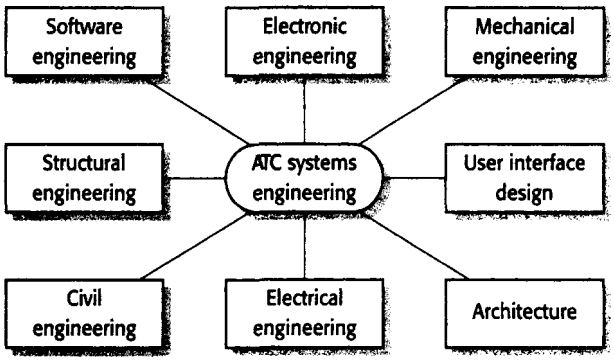


Figure 2.3 Disciplines involved in systems engineering

1. *Limited scope for rework during system development* Once some system engineering decisions, such as the siting of base stations in a mobile phone system, have been made, they are very expensive to change. Reworking the system design to solve these problems is rarely possible. One reason software has become so important in systems is that it allows changes to be made during system development, in response to new requirements.
2. *Interdisciplinary involvement* Many engineering disciplines may be involved in system engineering. There is a lot of scope for misunderstanding because different engineers use different terminology and conventions.

Systems engineering is an interdisciplinary activity involving teams drawn from various backgrounds. System engineering teams are needed because of the wide knowledge required to consider all the implications of system design decisions. As an illustration of this, Figure 2.3 shows some of the disciplines that may be involved in the system engineering team for an air traffic control (ATC) system that uses radars and other sensors to determine aircraft position.

For many systems, there are almost infinite possibilities for trade-offs between different types of sub-systems. Different disciplines negotiate to decide how functionality should be provided. Often there is no correct' decision on how a system should be decomposed. Rather, you may have several possible alternatives, but you may not be able to choose the best technical solution. Say one alternative in an air traffic control system is to build new radars rather than refit existing installations. If the civil engineers involved in this process do not have much other work, they may favour this alternative because it allows them to keep their jobs. They may then rationalise this choice with technical arguments.

2.2.1 System requirements definition

System requirements definitions specify what the system should do (its functions) and its essential and desirable system properties. As with software requirements analysis

(discussed in Part 2), creating system requirements definitions involves consultations with system customers and end-users. This requirements definition phase usually concentrates on deriving three types of requirement:

1. *Abstract functional requirements* The basic functions that the system must provide are defined at an abstract level. More detailed functional requirements specification takes place at the sub-system level. For example, in an air traffic control system, an abstract functional requirement would specify that a flight-plan database should be used to store the flight plans of all aircraft entering the controlled airspace. However, you would not normally specify the details of the database unless they affected the requirements of other sub-systems.
2. *System properties* These are non-functional emergent system properties such as availability, performance and safety, as I have discussed above. These non-functional system properties affect the requirements for all sub-systems.
3. *Characteristics that the system must not exhibit* It is sometimes as important to specify what the system must *not* do as it is to specify what the system should do. For example, if you are specifying an air traffic control system, you might specify that the system should not present the controller with too much information.

An important part of the requirements definition phase is to establish a set of overall objectives that the system should meet. These should not necessarily be expressed in terms of the system's functionality but should define why the system is being procured for a particular environment.

To illustrate what this means, say you are specifying a system for an office building to provide for fire protection and for intruder detection. A statement of objectives based on the system functionality might be:

> *To provide a fire and intruder alarm system for the building that will provide internal and external warning of fire or unauthorised intrusion.*

This objective states explicitly that there needs to be an alarm system that provides warnings of undesired events. Such a statement might be appropriate if you were replacing an existing alarm system. By contrast, a broader statement of objectives might be:

> *To ensure that the normal functioning of the work carried out in the building is not seriously disrupted by events such as fire and unauthorised intrusion.*

If you set out the objective like this, you both broaden and limit the design choices. For example, this objective allows for intruder protection using sophisticated locking technology—without any internal alarms. It may also exclude the use of sprinklers for fire protection because they can affect the building's electrical systems and so seriously disrupt work.

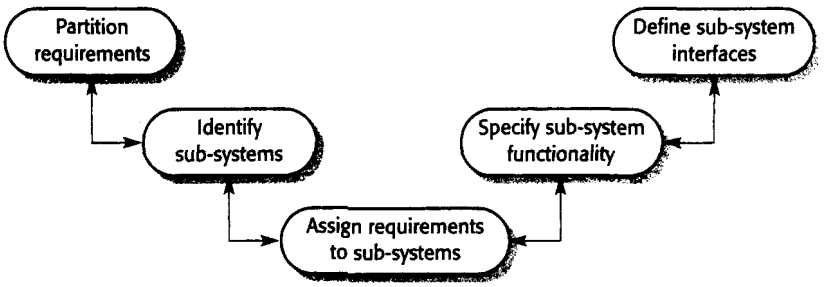


Figure 2.4 The system design process

A fundamental difficulty in establishing system requirements is that the problems that complex systems are usually built to help tackle are usually 'wicked problems' (Rittel and Webber, 1973). A 'wicked problem' is a problem that is so complex and where there are so many related entities that there is no definitive problem specification. The true nature of the problem emerges only as a solution is developed. An extreme example of a 'wicked problem' is earthquake planning. No one can accurately predict where the epicentre of an earthquake will be, what time it will occur or what effect it will have on the local environment. We cannot therefore completely specify how to deal with a major earthquake. The problem can only be tackled after it has happened.

2.2.2 System design

System design (Figure 2.4) is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are:

1. *Partition requirements* You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.
2. *Identify sub-systems* You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated. However, sub-system identification may also be influenced by other organisational or environmental factors.
3. *Assign requirements to sub-systems* You assign the requirements to sub-systems. In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems. Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.
4. *Specify sub-system functionality* You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase

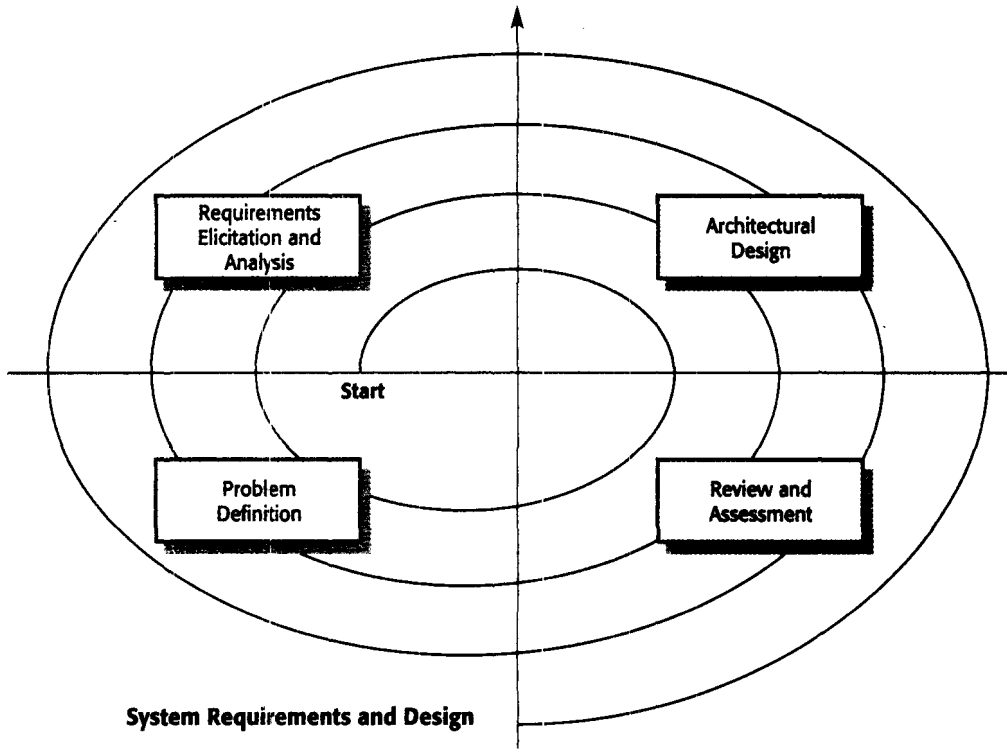


Figure 2.5 A spiral model of requirements and design

or, if the sub-system is a software system, part of the requirements specification activity for that system. You should also try to identify relationships between sub-systems at this stage.

5. *Define sub-system interfaces* You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

As the double-ended arrows in Figure 2.4 imply, there is a lot of feedback and iteration from one stage to another in this design process. As problems and questions arise, you often have to redo work done in earlier stages.

Although I have separated the processes of requirements engineering and design in this discussion, in practice they are inextricably linked. Constraints posed by existing systems may limit design choices, and these choices may be specified in the requirements. You may have to do some initial design to structure and organise the requirements engineering process. As the design process continues, you may discover problems with existing requirements and new requirements may emerge. Consequently, one way to think of these linked processes is as a spiral, as shown in Figure 2.5.

The spiral process reflects the reality that requirements affect design decisions and vice versa, and so it makes sense to interleave these processes. Starting in the

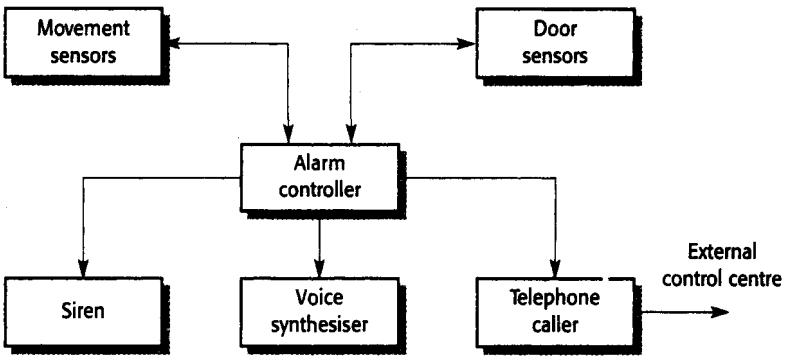


Figure 2.6 A simple burglar alarm system

centre, each round of the spiral may add detail to the requirements and the design. Some rounds may focus on requirements, some on design. Sometimes, new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

For almost all systems, there are many possible designs that meet the requirements. These cover a range of solutions that combine hardware, software and human operations. The solution that you chose for further development may be the most appropriate technical solution that meets the requirements. However, wider organisational and political considerations may influence the choice of solution. For example, a government client may prefer to use national rather than foreign suppliers for its system, even if the national product is technically inferior. These influences usually take effect in the review and assessment phase in the spiral model where designs and requirements may be accepted or rejected. The process ends when the review and evaluation shows that the requirements and high-level design are sufficiently detailed to allow the next phase of the process to begin.

2.2.3 System modelling

During the system requirements and design activity, systems may be modelled as a set of components and relationships between these components. These are normally illustrated graphically in a system architecture model that gives the reader an overview of the system organisation.

The system architecture may be presented as a block diagram showing the major sub-systems and the interconnections between these sub-systems. When drawing a block diagram, you should represent each sub-system using a rectangle, and you should show relationships between the sub-systems using arrows that link these rectangles. The relationships indicated may include data flow, a uses'/'used by' relationship or some other type of dependency relationship.

For example, Figure 2.6 shows the decomposition of an intruder alarm system into its principal components. The block diagram should be supplemented by brief descriptions of each sub-system, as shown in Figure 2.7.

Figure 2.7 Sub-system descriptions in the burglar alarm system

Sub-system	Description
Movement sensors	Detects movement in the rooms monitored by the system
Door sensors	Detects door opening in the external doors of the building
Alarm controller	Controls the operation of the system
Siren	Emits an audible warning when an intruder is suspected
Voice synthesiser	Synthesises a voice message giving the location of the suspected intruder
Telephone caller	Makes external calls to notify security, the police, etc.

At this level of detail, the system is decomposed into a set of interacting sub-systems. Each sub-system should be represented in a similar way until the system is decomposed into functional components. Functional components are components that, when viewed from the perspective of the sub-system, provide a single function. By contrast, a sub-system usually is multifunctional. Of course, when viewed from another perspective (say that of the component manufacturer), a functional component may itself be a system in its own right.

Historically, the system architecture model was used to identify hardware and software components that could be developed in parallel. However, this hardware/software distinction is becoming increasingly irrelevant. Almost all components now include some embedded computing capabilities. For example, a network linking machines will consist of physical cables plus repeaters and network gateways. The repeaters and the gateways include processors and software to drive these processors as well as specialised electronic components.

At the architectural level, it is now more appropriate to classify sub-systems according to their function before making decisions about hardware/software trade-offs. The decision to provide a function in hardware or software may be governed by non-technical factors such as the availability of off-the-shelf components or the time available to develop the component.

Block diagrams may be used for all sizes of system. Figure 2.8 shows the architecture of a much larger system for air traffic control. Several major sub-systems shown are themselves large systems. The arrowed lines that link these systems show information flow between these sub-systems.

2.2.4 Sub-system development

During sub-system development, the sub-systems identified during system design are implemented. This may involve starting another system engineering process for

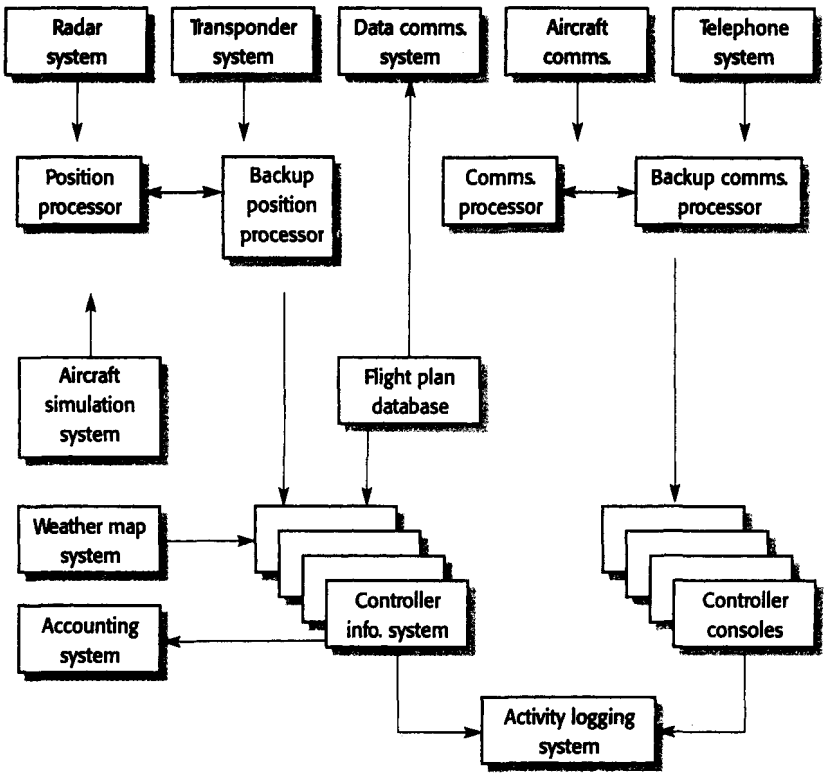


Figure 2.8 An architectural model of an air traffic control system

individual sub-systems or, if the sub-system is software, a software process involving requirements, design, implementation and testing.

Occasionally, all sub-systems are developed from scratch during the development process. Normally, however, some of the sub-systems are commercial, off-the-shelf (COTS) systems that are bought for integration into the system. It is usually much cheaper to buy existing products than to develop special-purpose components. At this stage, you may have to reenter the design activity to accommodate a bought-in component. COTS systems may not meet the requirements exactly but, if off-the-shelf products are available, it is usually worth the expense of rethinking the design.

Sub-systems are usually developed in parallel. When problems are encountered that cut across sub-system boundaries, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive. Often 'work-arounds' that compensate for the problem must be found. These 'work-arounds' usually involve software changes because of the software's inherent flexibility. This leads to changes in the software requirements so, as I have discussed in Chapter 1, it is important to design software for change so that the new requirements can be implemented without excessive additional costs.

2.2.5 Systems integration

During the systems integration process, you take the independently developed sub-systems and put them together to make up a complete system. Integration can be done using a 'big bang' approach, where all the sub-systems are integrated at the same time. However, for technical and managerial purposes, an incremental integration process where sub-systems are integrated one at a time is the best approach, for two reasons:

1. It is usually impossible to schedule the development of all the sub-systems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub-systems. When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interactions between the existing subsystems and the new sub-system.

Once the components have been integrated, an extensive programme of system testing takes place. This testing should be aimed at testing the interfaces between components and the behaviour of the system as a whole.

Sub-system faults that are a consequence of invalid assumptions about other sub-systems are often revealed during system integration. This may lead to disputes between the various contractors responsible for the different sub-systems. When problems are discovered in sub-system interaction, the contractors may argue about which sub-system is faulty. Negotiations on how to solve the problems can take weeks or months.

As more and more systems are built by integrating COTS hardware and software components, system integration is becoming increasingly important. In some cases, there is no separate sub-system development and the integration is, essentially, the implementation phase of the system.

2.2.6 System evolution

Large, complex systems have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. The system's computers are likely to be replaced with new, faster machines. The organisation that uses the system may reorganise itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system.

System evolution, like software evolution (discussed in Chapter 21), is inherently costly for several reasons:

1. Proposed changes have to be analysed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.

2. Because sub-systems are never completely independent, changes to one sub-system may adversely affect the performance or behaviour of other sub-systems. Consequent changes to these sub-systems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.

Systems that have evolved over time are often reliant on obsolete hardware and software technology. If they have a critical role in an organisation, they are known as *legacy systems*—systems that the organisation would like to replace but where the risks of introducing a new system are high. I discuss some issues with legacy systems in Section 2.4.

System decommissioning

System decommissioning means taking the system out of service after the end of its useful operational lifetime. For hardware systems this may involve disassembling and recycling materials or dealing with toxic substances. Software has no physical decommissioning problems, but some software may be incorporated in a system to assist with the decommissioning process. For example, software may be used to monitor the state of hardware components. When the system is decommissioned, components that are not worn can therefore be identified and reused in other systems.

If the data in the system that is being decommissioned is still valuable to your organisation, you may have to convert it for use by some other system. This can often involve significant costs as the data structures may be implicitly defined in the software itself. You have to analyse the software to discover how the data is structured and then write a program to reorganise the data into the required structures for the new system.

Organisations, people and computer systems

Socio-technical systems are enterprise systems that are intended to help deliver some organisational or business goal. This might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, etc. Because they are embedded in an organisational environment, the procurement, development and use of these system is influenced by the organisation's policies and procedures and by its working culture. The users of the system are people who are influenced by the way the

organisation is managed and by their interactions with other people inside and outside of the organisation.

Therefore, when you are trying to understand the requirements for a socio-technical system you need to understand its organisational environment. If you don't, the systems may not meet business needs, and users and their managers may reject the system.

Human and organisational factors from the system's environment that affect the system design include:

1. *Process changes* Does the system require changes to the work processes in the environment? If so, training will certainly be required. If changes are significant, or if they involve people losing their jobs, there is a danger that the users will resist the introduction of the system.
2. *Job changes* Does the system de-skill the users in an environment or cause them to change the way they work? If so, they may actively resist the introduction of the system into the organisation. Designs that involve managers having to change their way of working to fit the computer system are often resented. The managers may feel that their status in the organisation is being reduced by the system.
3. *Organisational changes* Does the system change the political power structure in an organisation? For example, if an organisation is dependent on a complex system, those who know how to operate the system have a great deal of political power.

These human, social and organisational factors are often critical in determining whether or not a system successfully meets its objectives. Unfortunately, predicting their effects on systems is very difficult for engineers who have little experience of social or cultural studies. To help understand the effects of systems on organisations, various methodologies have developed such as Mumford's socio-technics (Mumford, 1989) and Checkland's Soft Systems Methodology (Checkland and Scholes, 1990; Checkland, 1981). There have also been extensive sociological studies of the effects of computer-based systems on work (Ackroyd, et al., 1992).

Ideally, all relevant organisational knowledge should be included in the system specification so that the system designers may take it into account. In reality, this is impossible. System designers have to make assumptions based on other comparable systems and on common sense. If they get these wrong, the system may malfunction in unpredictable ways. For example, if the designers of a system do not understand that different parts of an organisation may actually have conflicting objectives, then any organisation-wide system that is developed will inevitably have some dissatisfied users.

2.3.1 Organisational processes

In Section 2.2, I introduced a system engineering process model that showed the sub-processes involved in system development. However, the development process is not the only process involved in systems engineering. It interacts with the

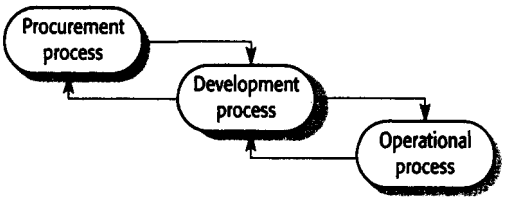


Figure 2.9 Procurement, development and operational processes

system procurement process and with the process of using and operating the system. This is illustrated in Figure 2.9.

The procurement process is normally embedded within the organisation that will buy and use the system (the client organisation). The process of system procurement is concerned with making decisions about the best way for an organisation to acquire a system and deciding on the best suppliers of that system.

Large complex systems usually consist of a mixture of off-the-shelf and specially built components. One reason why more and more software is included in systems is that it allows more use of existing hardware components, with the software acting as a 'glue' to make these hardware components work together effectively. The need to develop this 'glueware' is one reason why the savings from using off-the-shelf components are sometimes not as great as anticipated. I discuss COTS systems in more detail in Chapter 18.

Figure 2.10 shows the procurement process for both existing systems and systems that have to be specially designed. Some important points about the process shown in this diagram are:

1. Off-the-shelf components do not usually match requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing a system means that you have to find the closest match between the system requirements and the facilities offered by off-the-shelf systems. You may then have to modify the requirements and this can have knock-on effects on other sub-systems.
2. When a system is to be built specially, the specification of requirements acts as the basis of a contract for the system procurement. It is therefore a legal, as well as a technical, document.
3. After a contractor to build a system has been selected, there is a contract negotiation period where you may have to negotiate further changes to the requirements and discuss issues such as the cost of changes to the system.

I have already outlined the main phases of the system development process. Complex systems are usually developed by a different organization (the supplier) from the organization that is procuring the system. The reason for this is that the procurer's business is rarely system development so its employees do not have the

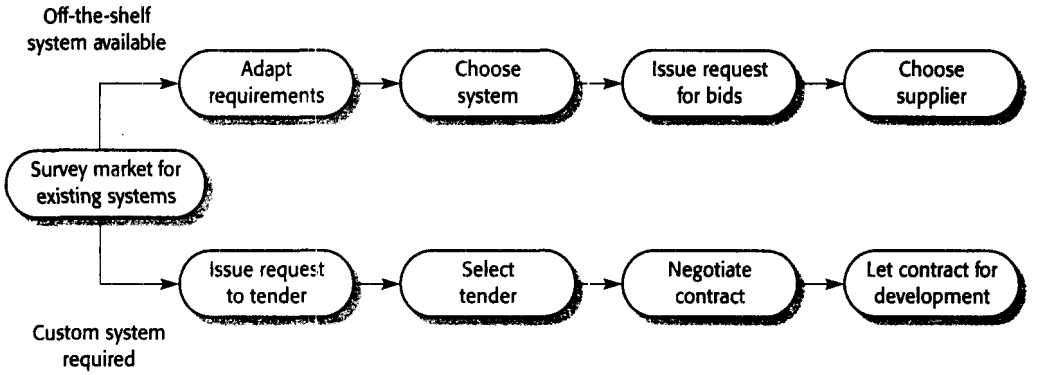


Figure 2.10 The system procurement process

skills needed to develop complex systems themselves. In fact, very few single organisations have the capabilities to design, manufacture and test all the components of a large, complex system.

This supplier, who is usually called the *principal contractor*, may contract out the development of different sub-systems to a number of sub-contractors. For large systems, such as air traffic control systems, a group of suppliers may form a consortium to bid for the contract. The consortium should include all of the capabilities required for this type of system, such as computer hardware suppliers, software developers, peripheral suppliers and suppliers of specialist equipment such as radars.

The procurer deals with the contractor rather than the sub-contractors so that there is a single procurer/supplier interface. The sub-contractors design and build parts of the system to a specification that is produced by the principal contractor. Once completed, the principal contractor integrates these different components and delivers them to the customer buying the system. Depending on the contract, the procurer may allow the principal contractor a free choice of sub-contractors or may require the principal contractor to choose sub-contractors from an approved list.

Operational processes are the processes that are involved in using the system for its defined purpose. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system. Undetected problems may arise at this stage because the system specification may contain errors or omissions. While the system may perform to specification, its functions may not meet the real operational needs. Consequently, the operators may not use the system as its designers intended.

The key benefit of having people in a system is that people have a unique capability of being able to respond effectively to unexpected situations even when they have never had direct experience of these situations. Therefore, when things go wrong,

the operators can often recover the situation, although this may sometimes mean that the defined process is violated. Operators also use their local knowledge to adapt and improve processes. Normally, the actual operational process is different from that anticipated by the system designers.

This means that designers should design operational processes to be flexible and adaptable. The operational processes should not be too constraining, they should not require operations to be done in a particular order, and the system software should not rely on a specific process being followed. Operators usually improve the process because they know what does and does not work in a real situation.

An issue that may only emerge after the system goes into operation is the problem of operating the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another. More subtle problems might arise because different systems have different user interfaces. Introducing the new system may increase the operator error rate for existing systems as the operators mix up user interface commands.

2.4 Legacy systems

Because of the time and effort required to develop a complex system, large computer-based systems usually have a long lifetime. For example, military systems are often designed for a 20-year lifetime, and much of the world's air traffic control still relies on software and operational processes that were originally developed in the 1960s and 1970s. It is sometimes too expensive and too risky to discard such business-critical systems after a few years of use. Their development continues throughout their life with changes to accommodate new requirements, new operating platforms, and so forth.

Legacy systems are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology. These systems include not only hardware and software but also legacy processes and procedures—old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components,

Legacy systems are often business-critical systems. They are maintained because it is too risky to replace them. For example, for most banks the customer accounting system was one of their earliest systems. Organisational policies and procedures may rely on this system. If the bank were to scrap and replace the customer accounting software (which may run on expensive mainframe hardware) then there would be a serious business risk if the replacement system didn't work properly. Furthermore, existing procedures would have to change, and this may upset the people in the organisation and cause difficulties with the bank's auditors.

Figure 2.11 illustrates the logical parts of a legacy system and their relationships:

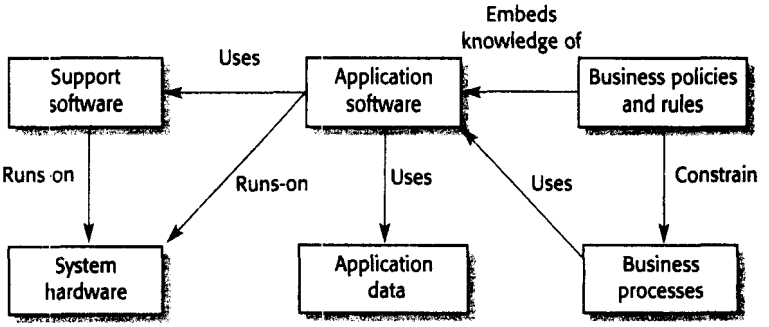


Figure 2.11 Legacy system components

1. *System hardware* In many cases, legacy systems have been written for mainframe hardware that is no longer available, that is expensive to maintain and that may not be compatible with current organisational IT purchasing policies.
2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
3. *Application software* The application system that provides the business services is usually composed of a number of separate programs that have been developed at different times. Sometimes the term *legacy system* means this application software system rather than the entire system.
4. *Application data* These are the data that are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent and may be duplicated in several files.
5. *Business processes* These are processes that are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.
6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 2.12. Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers.

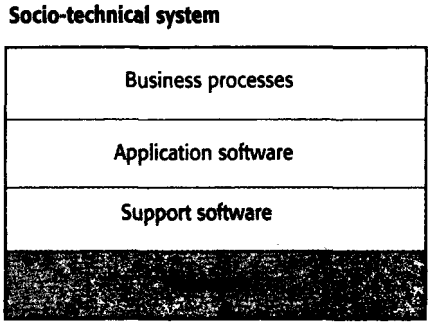


Figure 2.12 Layered model of a legacy system

In practice, this simple encapsulation rarely works, and changes to one layer of the system may require consequent changes to layers that are both above and below the changed level. The reasons for this are:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include

KEY POINTS

- Socio-technical systems include computer hardware, software and people, and are situated within an organisation. They are designed to help the organisation meet some broad goal.
- The emergent properties of a system are characteristic of the system as a whole rather than of its component parts. They include properties such as performance, reliability, usability, safety and security. The success or failure of a system is often dependent on these emergent properties.
- The systems engineering process includes specification, design, development, integration and testing. System integration, where sub-systems from more than one supplier must be made to work together, is particularly critical.
- Human and organisational factors such as organisational structure and politics have a significant effect on the operation of socio-technical systems.
- Within an organisation, there are complex interactions between the processes of system procurement, development and operation.
- A legacy system is an old system that still provides essential business services.
- Legacy systems are not just application software systems. They are socio-technical systems so include business processes, application software, support software and system hardware.