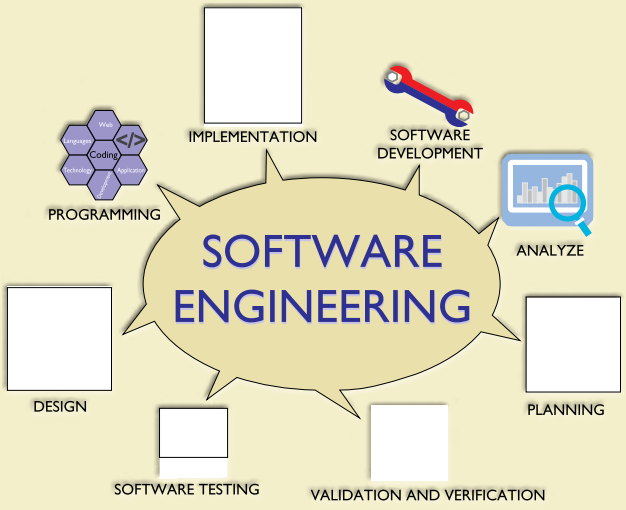
CHAPTER

1

INTRODUCTION



CHAPTER OUTLINE

- Evolution of technology with time
- Relative changes of hardware and software costs over time
- Exploratory program development
- Increase in development time and effort with problem size
- Human cognition mechanism model
- Data flow model of a car assembly plant
- Evolution of software design techniques
- Computer systems engineering

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

—IEEE Standard Glossary of
Software Engineering Terminology

LEARNING OBJECTIVES

- Exploratory style of software development and its shortcomings.
- Perceived problem complexity and techniques to overcome them
- Evolution of software engineering principles

Commercial usage of computers now spans the last sixty years. Computers were very slow in the initial years and lacked sophistication. Since then, their computational power and sophistication increased rapidly, while their prices dropped dramatically. To get an idea of the kind of improvements that have occurred to computers, consider the following analogy. If similar improvements could have occurred to aircrafts, now personal mini-airplanes should have become available, costing as much as a bicycle, and flying at over 1000 times the speed of the supersonic jets. To say it in other words, the rapid strides in computing technologies are unparalleled in any other field of human endeavour.

Let us now reflect the impact of the astounding progress made to the hardware technologies on the software. The more powerful a computer is, the more sophisticated programs can it run. Therefore, with every increase in the raw computing capabilities of computers, software engineers have been called upon to solve increasingly larger and complex problems, and that too in cost-effective and efficient ways. Software engineers have coped up with this challenge by innovating and building upon their past programming experiences.

The innovations and past experiences towards writing good quality programs cost-effectively, have contributed to the emergence of the software engineering discipline.

Let us now examine the scope of the software engineering discipline more closely.

What is software engineering?

A popular definition of software engineering is: "A systematic collection of good program development practices and techniques". Good program development techniques have resulted from research innovations as well as from the lessons learnt by programmers through years of programming experiences. An alternative definition of software engineering is: "An *engineering approach* to develop software".

Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Let us now try to figure out what exactly is meant by an *engineering approach* to develop software. We explain this using an analogy. Suppose you have asked a petty contractor to build a small house for you. Petty contractors are not really experts in house building. They normally carry out minor repair works

and at most undertake very small building works such as the construction of boundary walls. Now faced with the task of building a complete house, your petty contractor would draw upon all his knowledge regarding house building. Yet, he may often be clueless regarding what to do. For example, he might not know the optimal proportion in which cement and sand should be mixed to realise sufficient strength for supporting the roof. In such situations, he would have to fall back upon his intuitions. He would normally succeed in his work, if the house you asked him to construct is sufficiently small. Of course, the house constructed by him may not look as good as one constructed by a professional, may lack proper planning, and display several defects and imperfections. It may even cost more and take longer to build.

Now, suppose you entrust your petty contractor to build a large 50-storeyed commercial complex for you. He might exercise prudence, and politely refuse to undertake your request. On the other hand, he might be ambitious and agree to undertake the task. In the later case, he is sure to fail. The failure might come in several forms—the building might collapse during the construction stage itself due to his ignorance of the basic theories concerning the strengths of materials; the construction might get unduly delayed, since he may not prepare proper estimates and detailed plans regarding the types and quantities of raw materials required, the times at which these are required, etc. In short, to be successful in constructing a building of large magnitude, one needs a good understanding of various civil and architectural engineering techniques such as analysis, estimation, prototyping, planning, designing, and testing. Similar is the case with the software development projects. For sufficiently small-sized problems, one might proceed according to one's intuition and succeed; though the solution may have several imperfections, cost more, take longer to complete, etc. But, failure is almost certain, if one without a sound understanding of the software engineering principles undertakes a large-scale software development work.

Is software engineering a science or an art?

Several people hold the opinion that writing good quality programs is an art. In this context, let us examine whether software engineering is really a form of art or is it akin to other engineering disciplines. There exist several fundamental issues that set engineering disciplines such as software engineering and civil engineering apart from both science and arts disciplines. Let us now examine where software engineering stands based on an investigation into these issues:

- Just as any other engineering discipline, software engineering makes heavy use of the knowledge that has accrued from the experiences of a large number of practitioners. These past experiences have been systematically organised and wherever possible theoretical basis to the empirical observations have been provided. Whenever no reasonable theoretical justification could be provided, the past experiences have been adopted as rule of thumb. In contrast, all scientific solutions are constructed through rigorous application of provable principles.
- As is usual in all engineering disciplines, in software engineering several conflicting goals are encountered while solving a problem. In such situations, several alternate solutions are first proposed. An appropriate solution is chosen out of the candidate solutions based on various trade-offs that made on account of issues such as cost,

maintainability, and usability. Therefore, while arriving at the final solution, several iterations and backtracking are usually needed. In science, on the other hand, only unique solutions are possible.

- Engineering disciplines such as software engineering make use of only well-understood and well-documented principles. Art, on the other hand, is often based on making subjective judgement based on qualitative attributes and using ill-understood principles.

From the above discussions, we can easily infer that software engineering is in many ways similar to other engineering disciplines such as civil engineering or electronics engineering.

1.1 EVOLUTION—FROM AN ART FORM TO AN ENGINEERING DISCIPLINE

In this section, we review how starting from an esoteric art form, the software engineering discipline has evolved over the years.

1.1.1 Evolution of an Art into an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.

The early programmers used an *ad hoc* programming style. This style of program development is now variously being referred to as *exploratory*, *build and fix*, and *code and fix* styles.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies. The exploratory style comes naturally to all first time programmers. Later in this chapter, we point out that except for trivial software development problems, the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

In a build and fix style, a program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed are fixed.

As we have already pointed out, the build and fix style was widely adopted by the programmers in the early years of computing history. We can consider the exploratory program development style as an art—since this style, as is the case with any art, is mostly guided by intuition. There are many stories about programmers in the past who were like proficient artists and could write good programs using an essentially build and fix model and some esoteric knowledge. The bad programmers were left to wonder how could some programmers effortlessly write elegant and correct programs each time. In contrast, the programmers working in modern software industry rarely make use of any esoteric knowledge and develop software by applying some well-understood principles.

1.1.2 Evolution Pattern for Engineering Disciplines

If we analyse the evolution of the software development styles over the last sixty years,

we can easily notice that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. As a matter of fact, this pattern of evolution is not very different from that seen in other engineering disciplines. Irrespective of whether it is iron making, paper making, software development, or building construction; evolution of technology has followed strikingly similar patterns. This pattern of technology development has schematically been shown in Figure 1.1. It can be seen from Figure 1.1 that every technology in the initial years starts as a form of art. Over time, it graduates to a craft and finally emerges as an engineering discipline. Let us illustrate this fact using an example. Consider the evolution of the iron making technology. In ancient times, only a few people knew how to make iron. Those who knew iron making, kept it a closely-guarded secret. This esoteric knowledge got transferred from generation to generation as a family secret. Slowly, over time technology graduated from an art to a craft form where tradesmen shared their knowledge with their apprentices and the knowledge pool continued to grow. Much later, through a systematic organisation and documentation of knowledge, and incorporation of scientific basis, modern steel making technology emerged. The story of the evolution of the software engineering discipline is not much different. As we have already pointed out, in the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were discovered by programmers along with research innovations have systematically been organised into a body of knowledge that forms the discipline of software engineering.

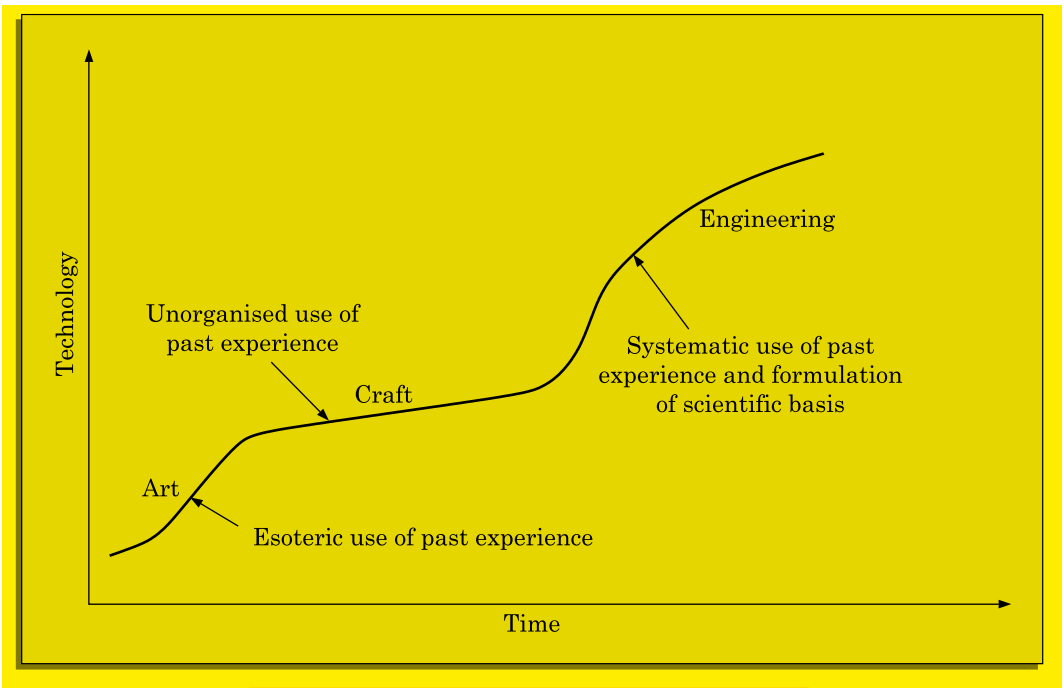


FIGURE 1.1 Evolution of technology with time.

Software engineering principles are now being widely used in industry, and new principles are still continuing to emerge at a very rapid rate—making this discipline highly dynamic. In spite of its wide acceptance, critics point out that many of the methodologies and guidelines provided by the software engineering discipline lack scientific basis, are subjective, and often inadequate. Yet, there is no denying the fact that adopting software engineering techniques facilitates development of high quality software in a cost-effective and timely manner. Software engineering practices have proven to be indispensable to the development of large software products—though exploratory styles are often used successfully to develop small programs such as those written by students as classroom assignments.

1.1.3 A Solution to the Software Crisis

At present, software engineering appears to be among the few options that are available to tackle the present software crisis. But, what exactly is the present software crisis? What are its symptoms, causes, and possible solutions? To understand the present software crisis, consider the following facts. The expenses that organisations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years (see Figure 1.2). As can be seen in the figure, organisations are spending increasingly larger portions of their budget on software as compared to that on hardware. Among all the symptoms of the present software crisis, the trend of increasing software costs is probably the most vexing.

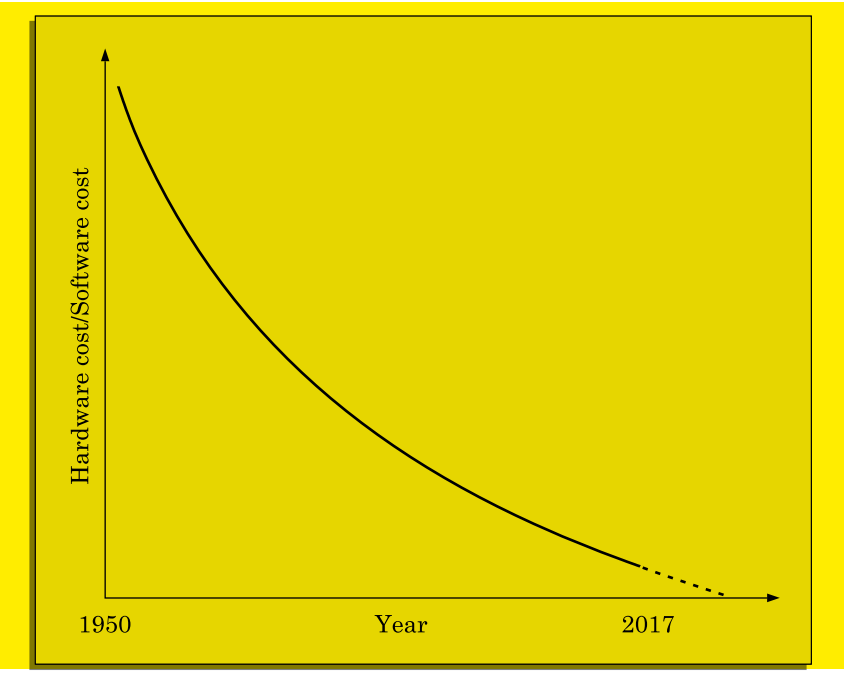


FIGURE 1.2 Relative changes of hardware and software costs over time.

At present, many organisations are actually spending much more on software than on hardware. If this trend continues, we might soon have a rather amusing scenario. Not

long ago, when you bought any hardware product, the essential software that ran on it came free with it. But, unless some sort of revolution happens, in not very distant future, hardware prices would become insignificant compared to software prices—when you buy any software product the hardware on which the software runs would come free with the software!!!

> Not only are the software products becoming progressively more expensive than hardware, but they also present a host of other problems to the customers—software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late.

The symptoms of software crisis are not hard to observe. But, what are the factors that have contributed to the present software crisis? Apparently, there are many factors, the important ones being—rapidly increasing problem size, lack of adequate training in software engineering techniques, increasing skill shortage, and low productivity improvements. What is the remedy? It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, coupled with further advancements to the software engineering discipline itself.

With this brief discussion on the evolution and impact of the discipline of software engineering, we now examine some basic concepts pertaining to the different types of software development projects that are undertaken by software companies.

1.2 SOFTWARE DEVELOPMENT PROJECTS

Before discussing about the various types of development projects that are being undertaken by software development companies, let us first understand the important ways in which professional software differs from toy software such as those written by a student in his fi Program gnment.

1.2.1 Programs *versus* Products

Many toy software are being developed by individuals such as students for their classroom assignments and hobbyists for their personal use. These are usually small in size and support limited functionalities. Further, the author of a program is usually the sole user of the software and himself maintains the code. These toy software therefore usually lack good user-interface and proper documentation. Besides these may have poor maintainability, efficiency, and reliability. Since these toy software do not have any supporting documents such as users' manual, maintenance manual, design document, test documents, etc., we call these toy software as *programs*.

Product

In contrast, professional software usually have multiple users and, therefore, have good user-interface, proper users' manuals, and good documentation support. Since, a software product has a large number of users, it is systematically designed, carefully implemented, and thoroughly tested. In addition, a professionally written software usually consists not only of the program code but also of all associated documents such as requirements specification document, design document, test document, users' manuals, etc. A further difference is that professional software are often too large and complex to be developed by any single individual. It is usually developed by a group of developers working in a team.

A professional software is developed by a group of software developers working together in a team. It is therefore necessary for them to use some systematic development methodology. Otherwise, they would find it very difficult to interface and understand each other's work, and produce a coherent set of documents.

Even though software engineering principles are primarily intended for use in development of professional software, many results of software engineering can effectively be used for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles is often not worthwhile. An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. CAR Hoare [1994] observed that rigorously using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castles for children to play.

1.2.2 Types of Software Development Projects

A software development company typically has a large number of on-going projects. Each of these projects may be classified into software product development projects or services type of projects. These two broad classes of software projects can be further classified into subclasses as shown in Figure 1.3

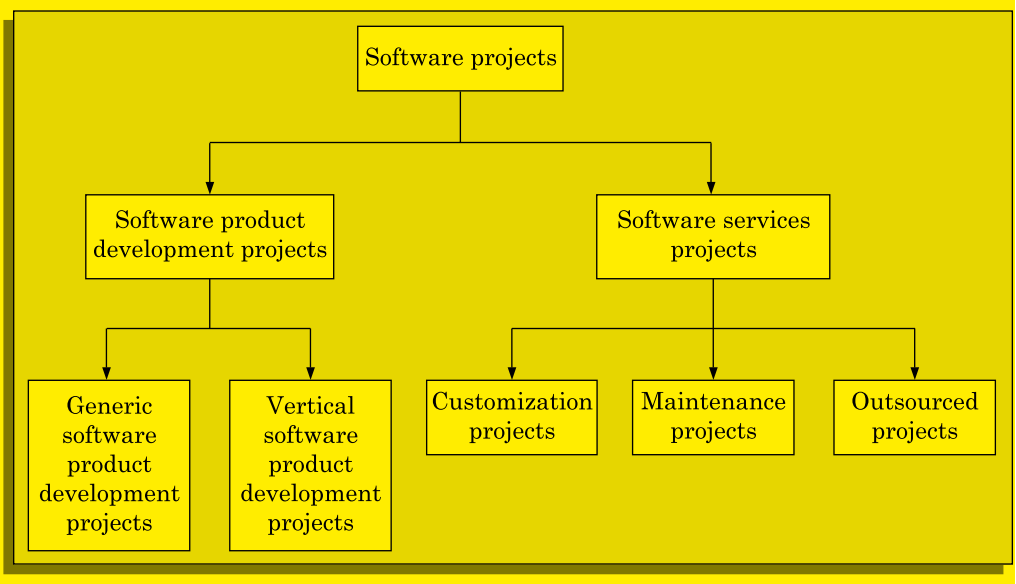


FIGURE 1.3 A classification of software projects.

A software product development project may be either to develop a generic product or a domain specific product. A generic software product development project concerns about developing a software that would be sold to a large number of customers. Since a generic software product is sold to a broad spectrum of customers, it is said to have a horizontal market. On the other hand, the services projects may either involve customizing some existing software, maintaining or developing some outsourced software. Since a specific segment of customers are targeted, these software products are said to have a vertical market. In the following, we distinguish between these two major types of software projects.

Software products

We all know of a variety of software such as Microsoft's Windows operating system and Office suite, and Oracle Corporation's Oracle 8i database management software. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called *generic software products* since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own. Of course, it may base its design discretion on feedbacks collected from a large number of users. Typically, each software product is targetted to some market segment (set of users). Many companies find it advantageous to develop *product lines* that target slightly different market segments based on variations of essentially the same software. For example, Microsoft targets desktops and laptops through its *Windows* 8 operating system, while it targets high-end mobile handsets through its *Windows mobile* operating system, and targets servers through its *Windows server* operating system.

In contrast to the generic products, domain specific software products are sold to specific categories of customers and are said to have a vertical market. Domain specific software products target specific segments of customers (called *verticals*) such as banking, telecommunication, finance and accounts, and medical. Examples of domain specific software products are BANCS from TCS and FINACLE from Infosys in the banking domain and AspenPlus from Aspen Corporation in the chemical process simulation.

Software services

Software services covers a large gamut of software projects such as customization, outsourcing, maintenance, testing, and consultancy. At present, there is a rapid growth in the number of software services projects that are being undertaken world-wide and software services are poised to become the dominant type of software projects. One of the reasons behind this situation is the steep growth in the available code base. Over the past few decades, a large number of programs have already been developed. Available programs can therefore be modified to quickly fulfil the specific requirements of any customer. At present, there is hardly any software project in which the program code is written from scratch, and software is being mostly developed by customizing some existing software. For example, to develop a software to automate the payroll generation activities of an educational institute, the vendor may customize an existing software that might has been developed earlier for a different client or educational institute. Due to heavy reuse of code, it has now become possible to develop even large software systems in rather short periods of time. Therefore, typical project durations are at present only a couple of months and multi-year projects have become very rare.

Development of *outsourced software* is a type of software service. Outsourced software projects may arise for many reasons. Sometimes, it can make good commercial sense for a company developing a large project to outsource some parts of its development work to other companies. The reasons behind such a decision may be many. For example, a company might consider the outsourcing option, if it feels that it does not have sufficient expertise to develop some specific parts of the software; or it may determine that some parts can be developed cost-effectively by another company. Since an outsourced project is a small

part of some larger project, outsourced projects are usually small in size and need to be completed within a few months or a few weeks of time.

The types of development projects that are being undertaken by a company can have an impact on its profitability. For example, a company that has developed a generic software product usually gets an uninterrupted stream of revenue spread over several years. However, development of a generic software product entails substantial upfront investment. Further, any return on this investment is subject to the risk of customer acceptance. On the other hand, outsourced projects are usually less risky, but fetch only one time revenue to the developing company.

1.2.3 Software Projects Being Undertaken by Indian Companies

Indian software companies have excelled in executing software services projects and have made a name for themselves all over the world. Of late, the Indian companies have slowly started to focus on product development as well. Can you recall the names of a few software products developed by Indian software companies? Let us try to hypothesise the reason for this situation. Generic product development entails certain amount of business risk. A company needs to invest upfront and there is substantial risks concerning whether the investments would turn profitable. Possibly, the Indian companies were risk averse.

Till recently, the world-wide sales revenue of software products and services were evenly matched. But, of late the services segment has been growing at a faster pace due to the advent of application service provisioning and cloud computing. We discuss these issues in Chapter 15.

1.3 EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT

We have already discussed that the *exploratory program development style* refers to an informal development style where the programmer makes use of his own intuition to develop a program rather than making use of the systematic body of knowledge categorized under the software engineering discipline. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software. Though the exploratory style imposes no rules a typical development starts after an initial briefing from the customer. Based on this briefing, the developers start coding to develop a working program. The software is tested and the bugs found are fixed. This cycle of testing and bug fixing continues till the software works satisfactorily for the customer. A schematic of this work sequence in a build and fix style has been shown graphically in Figure 1.4. Observe that coding starts after an initial customer briefing

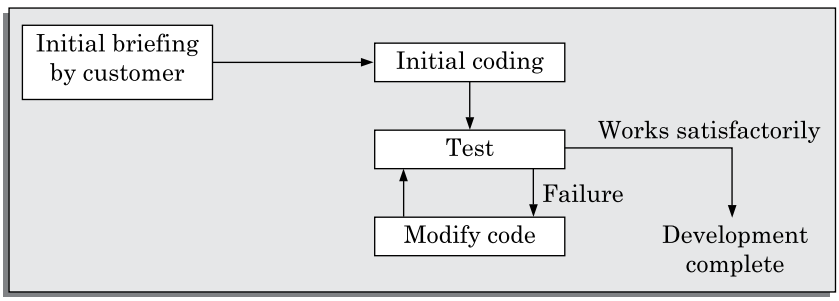


FIGURE 1.4 Exploratory program development.

about what is required. After the program development is complete, a test and fix cycle continues till the program becomes acceptable to the customer.

An exploratory development style can be successful when used for developing very small programs, and not for professional software. We had examined this issue with the help of the petty contractor analogy. Now let us examine this issue more carefully.

What is wrong with the exploratory style of software development?

Though the exploratory software development style is intuitively obvious, no software team can remain competitive if it uses this style of software development. Let us investigate the reasons behind this. In an exploratory development scenario, let us examine how do the effort and time required to develop a professional software increases with the increase in program size. Let us first consider that exploratory style is being used to develop a professional software. The increase in development effort and time with problem size has been indicated in Figure 1.5. Observe the thick line plot that represents the case in which the exploratory style is used to develop a program. It can be seen that as the program size increases, the required effort and time increases almost exponentially. For large problems, it would take too long and cost too much to be practically meaningful to develop the program using the exploratory style of development. The exploratory development approach is said to break down after the size of the program to be developed increases beyond certain value. For example, using the exploratory style, you may easily solve a problem that requires writing only 1000 or 2000 lines of source code. But, if you are asked to solve a problem that would require writing one million lines of source code, you may never be able to complete it using the exploratory style; irrespective of the amount time or effort you might invest to solve it. Now observe the thin solid line plot in Figure 1.5 which represents the case when development is carried out using software engineering principles. In this case, it becomes possible to solve a problem with effort and time that is almost linear in program size. On the other hand, if programs could be written automatically by machines, then the increase in effort and time with size would be even closer to a linear (dotted line plot) increase with size.

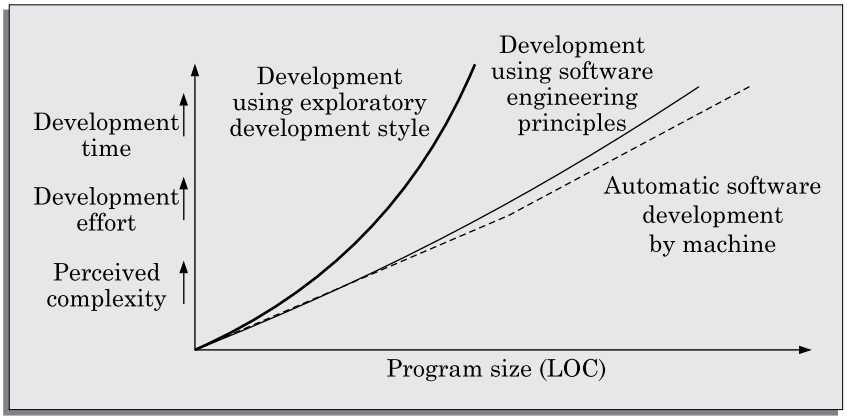


FIGURE 1.5 Increase in development time and effort with problem size.

Now let us try to understand why does the effort required to develop a program grow exponentially with program size when the exploratory style is used and then this

approach to develop a program completely breaks down when the program size becomes large? To get an insight into the answer to this question, we need to have some knowledge of the human cognitive limitations (see the discussion on human psychology in subsection 1.3.1). As we shall see, the perceived (or psychological) complexity of a problem grows exponentially with its size. Please note that the perceived complexity of a problem is not related to the time or space complexity issues with which you are likely to be familiar with from a basic course on algorithms.

Even if the exploratory style causes the perceived difficulty of a problem to grow exponentially due to human cognitive limitations, how do the software engineering principles help to contain this exponential rise in complexity with problem size and hold it down to almost a linear increase? We will discuss in subsection 1.3.2 that software engineering principle help achieve this by profusely making use of the abstraction and decomposition techniques to overcome the human cognitive limitations. You may still wonder that when software engineering principles are used, why does the curve not become completely linear? The answer is that it is very difficult to apply the decomposition and abstraction principles to completely overcome the problem complexity.

The psychological or perceived complexity of a problem concerns the difficulty level experienced by a programmer while solving the problem using the exploratory development style.

Summary of the shortcomings of the exploratory style of software development: We briefly summarise the important shortcomings of using the exploratory development style to develop a professional software:

- The foremost difficulty is the exponential growth of development time and effort with problem size and large-sized software becomes almost impossible using this style of development.
- The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design would result in highly unstructured and poor quality code.
- It becomes very difficult to use the exploratory style in a team development environment. In the exploratory style, the development work is undertaken without any proper design and documentation. Therefore it becomes very difficult to meaningfully partition the work among a set of developers who can work concurrently. On the other hand, team development is indispensable for developing modern software—most software mandate huge development efforts, necessitating team effort for developing these. Besides poor quality code, lack of proper documentation makes any later maintenance of the code very difficult.

1.3.1 Perceived Problem Complexity: An Interpretation Based on Human Cognition Mechanism

The rapid increase of the perceived complexity of a problem with increase in problem size can be explained from an interpretation of the human cognition mechanism. A simple understanding of the human cognitive mechanism would also give us an insight into why the exploratory style of development leads to an undue increase in the time and effort required to develop a programming solution. It can also explain why it becomes practically infeasible to solve problems larger than a certain size while using an

exploratory style; whereas using software engineering principles, the required effort grows almost linearly with size (as indicated by the thin solid line in Figure 1.5).

Psychologists say that the human memory can be thought to consist of two distinct parts [Miller, 1956]: short-term and long-term memories. A schematic representation of these two types of memories and their roles in human cognition mechanism has been shown in Figure 1.6. In Figure 1.6, the block labelled sensory organs represents the five human senses sight, hearing, touch, smell, and taste. The block labelled actuator represents neuromotor organs such as hand, finger, feet, etc. We now elaborate this human cognition model in the following subsection.

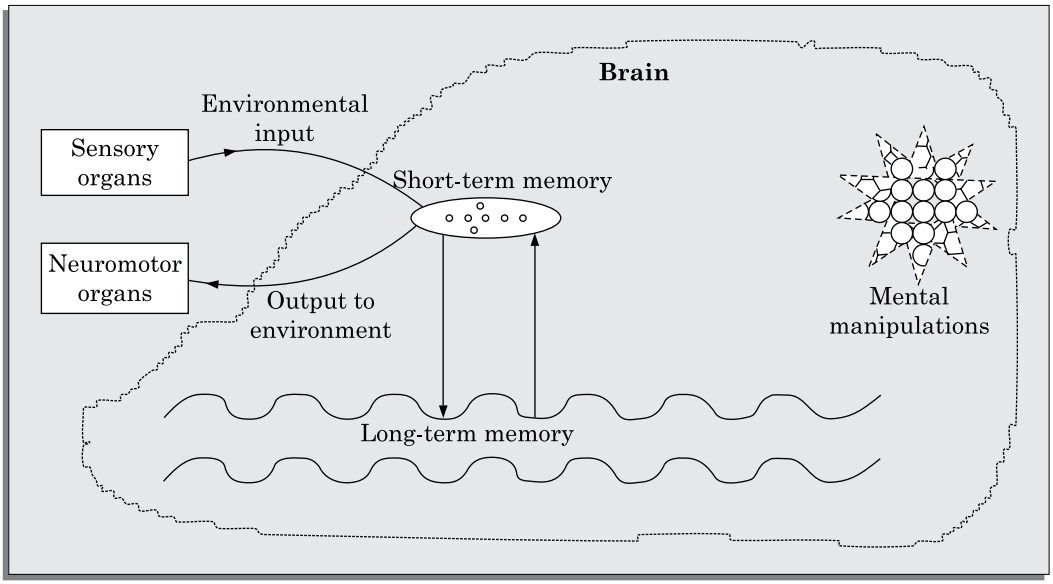


FIGURE 1.6 Human cognition mechanism model.

Short-term memory: The short-term memory, as the name itself suggests, can store information for a short while—usually up to a few seconds, and at most for a few minutes. The short-term memory is also sometimes referred to as the *working memory*. The information stored in the short-term memory is immediately accessible for processing by the brain. The short-term memory of an average person can store up to seven items; but in extreme cases it can vary anywhere from five to nine items (7 ± 2). As shown in Figure 1.6, the short-term memory participates in all interactions of the human mind with its environment.

It should be clear that the short-term memory plays a very crucial part in the human cognition mechanism. All information collected through the sensory organs are first stored in the short-term memory. The short-term memory is also used by the brain to drive the neuromotor organs. The mental manipulation unit also gets its inputs from the short-term memory and stores back any output it produces. Further, information retrieved from the long-term memory first gets stored in the short-term memory. For example, if you are asked the question: "If it is 10AM now, how many hours are remaining today?" First, 10AM would be stored in the short-term memory. Next, the information that a day

is 24 hours long would be fetched from the long-term memory into the short-term memory. The mental manipulation unit would compute the difference (24 – 10), and 14 hours would get stored in the short-term memory. As you can notice, this model is very similar to the organisation of a computer in terms of cache, main memory, and processor.

An item stored in the short-term memory can get lost either due to decay with time or displacement by newer information. This restricts the duration for which an item is stored in the short-term memory to few tens of seconds. However, an item can be retained longer in the short-term memory by recycling. That is, when we repeat or refresh an item consciously, we can remember it for a much longer duration. Certain information stored in the short-term memory, under certain circumstances gets stored in the long-term memory.

Long-term memory: Unlike the short-term memory, the size of the long-term memory is not known to have a definite upper bound. The size of the long-term memory can vary from several million items to several billion items, largely depending on how actively a person exercises his mental faculty. An item once stored in the long-term memory, is usually retained for several years. But, how do items get stored in the long-term memory? Items present in the short-term memory can get stored in the long-term memory either through large number of refreshments (repetitions) or by forming links with already existing items in the long-term memory. For example, you possibly remember your own telephone number because you might have repeated (refreshed) it for a large number of times in your short-term memory. Let us now take an example of a situation where you may form links to existing items in the long-term memory to remember certain information. Suppose you want to remember the 10 digit mobile number 9433795369. To remember it by rote may be intimidating. But, suppose you consider the number as split into 9433 7953 69 and notice that 94 is the code for BSNL, 33 is the code for Kolkata, suppose 79 is your year of birth, and 53 is your roll number, and the rest of the two numbers are each one less than the corresponding digits of the previous number; you have effectively established links with already stored items, making it easier to remember the number.

Item: We have so far only mentioned the number of items that the long-term and the short-term memories can store. But, what exactly is an item? An *item* is any set of related information. According to this definition, a character such as *a* or a digit such as '5' can each be considered as an item. A word, a sentence, a story, or even a picture can each be a single item. Each item normally occupies one place in memory. The definition of an item as any set of related information implies that when you are able to establish some simple relationship between several different items, the information that should normally occupy several places can be stored using only one place in the memory. This phenomenon of forming one item from several items is referred to as *chunking* by psychologists. For example, if you are given the binary number 110010101001—it may prove very hard for you to understand and remember. But, the octal form of the number 6251 (i.e., the representation (110)(010)(101)(001)) may be much easier to understand and remember since we have managed to create chunks of three items each.

Evidence of short-term memory: Evidences of short-term memory manifest themselves in many of our day-to-day experiences. As an example of the short-term memory, consider the following situation. Suppose, you look up a number from the telephone directory

and start dialling it. If you find the number to be busy, you would dial the number again after a few seconds—in this case, you would be able to do so almost effortlessly without having to look up the directory. But, after several hours or days since you dialled the number last, you may not remember the number at all, and would need to consult the directory again.

The magical number 7: Miller called the number seven as the *magical number* [Miller, 1956] since if a person deals with seven or less number of unrelated information at a time these would be easily accommodated in the short-term memory. So, he can easily understand it. As the number of items that one has to deal with increases beyond seven, it becomes exceedingly difficult to understand it. This observation can easily be extended to writing programs.

A small program having just a few variables is within the easy grasp of an individual. As the number of independent variables in the program increases, it quickly exceeds the grasping power of an individual and would require an unduly large effort to master the problem. This outlines a possible reason behind the exponential nature of the effort-size plot (thick line) shown in Figure 1.5. Please note that the situation depicted in Figure 1.5 arises mostly due to the human cognitive limitations. Instead of a human, if a machine could be writing (generating) a program, the slope of the curve would be linear, as the cache size (short-term memory) of a computer is quite large. But, why does the effort-size curve become almost linear when software engineering principles are deployed? This is because software engineering principles extensively use the techniques that are designed specifically to overcome the human cognitive limitations. We discuss this issue in the next subsection.

When the number of details (or variables) that one has to track to solve a problem increases beyond seven, it exceeds the capacity of the short-term memory and it becomes exceedingly more difficult for a human mind to grasp the problem.

1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

We shall see throughout this book that a central theme of most of software engineering principles is the use of techniques to effectively tackle the problems that arise due to human cognitive limitations.

In the following subsections, with the help of Figure 1.7(a) and (b), we explain the essence of these two important principles and how they help to overcome the human cognitive limitations. In the rest of this book, we shall time and again encounter the use of these two fundamental principles in various forms and flavours in the different software development activities. A thorough understanding of these two principles is therefore needed.

Two important principles that are deployed by software engineering to overcome the problems arising due to human cognitive limitations are—abstraction and decomposition.

Abstraction

Abstraction refers to construction of a simpler version of a problem by ignoring the details. The principle of constructing an abstraction is popularly known as *modelling* (or *model construction*).

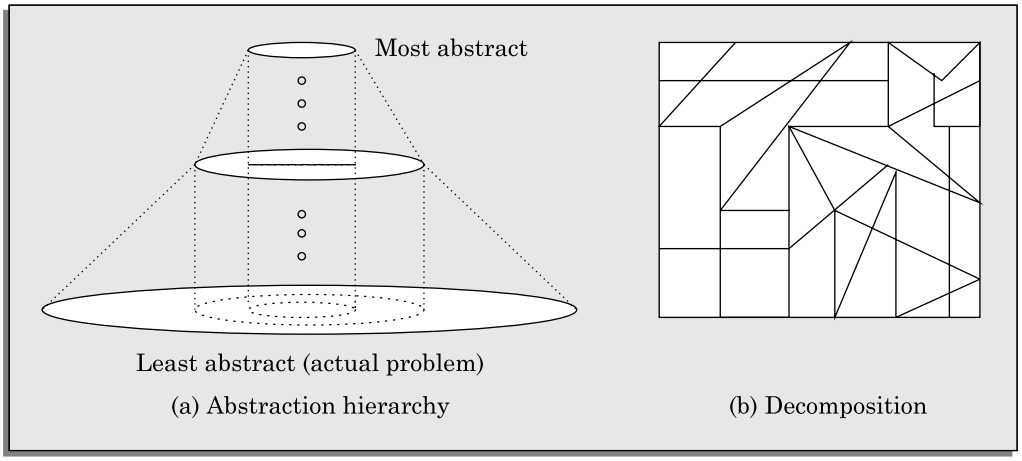


FIGURE 1.7 Schematic representation.

When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest. Whenever we omit some details of a problem to construct an abstraction, we construct a *model* of the problem. In every day life, we use the principle of abstraction frequently to understand a problem or to assess a situation. Consider the following two examples.

Abstraction is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.

- Suppose you are asked to develop an overall understanding of some country. No one in his right mind would start this task by meeting all the citizens of the country, visiting every house, and examining every tree of the country, etc. You would probably take the help of several types of abstractions to do this. You would possibly start by referring to and understanding various types of maps for that country. A map, in fact, is an abstract representation of a country. It ignores detailed information such as the specific persons who inhabit it, houses, schools, play grounds, trees, etc. Again, there are two important types of maps—physical and political maps. A *physical map* shows the physical features of an area; such as mountains, lakes, rivers, coastlines, and so on. On the other hand, the *political map* shows states, capitals, and national boundaries, etc. The physical map is an abstract model of the country and ignores the state and district boundaries. The political map, on the other hand, is another abstraction of the country that ignores the physical characteristics such as elevation of lands, vegetation, etc. It can be seen that, for the same object (e.g. country), several abstractions are possible. In each abstraction, some aspects of the object is ignored. We understand a problem by abstracting out different aspects of a problem (constructing different types of models) and understanding them. It is not very difficult to realise that proper use of the principle of abstraction can be a very effective help to master even intimidating problems.
- Consider the following situation. Suppose you are asked to develop an understanding of all the living beings inhabiting the earth. If you use the naive approach, you would start taking up one living being after another who inhabit the earth and

start understanding them. Even after putting in tremendous effort, you would make little progress and left confused since there are billions of living things on earth and the information would be just too much for any one to handle. Instead, what can be done is to build and understand an abstraction hierarchy of all living beings as shown in Figure 1.8. At the top level, we understand that there are essentially three fundamentally different types of living beings—plants, animals, and fungi. Slowly more details are added about each type at each successive level, until we reach the level of the different species at the leaf level of the abstraction tree.

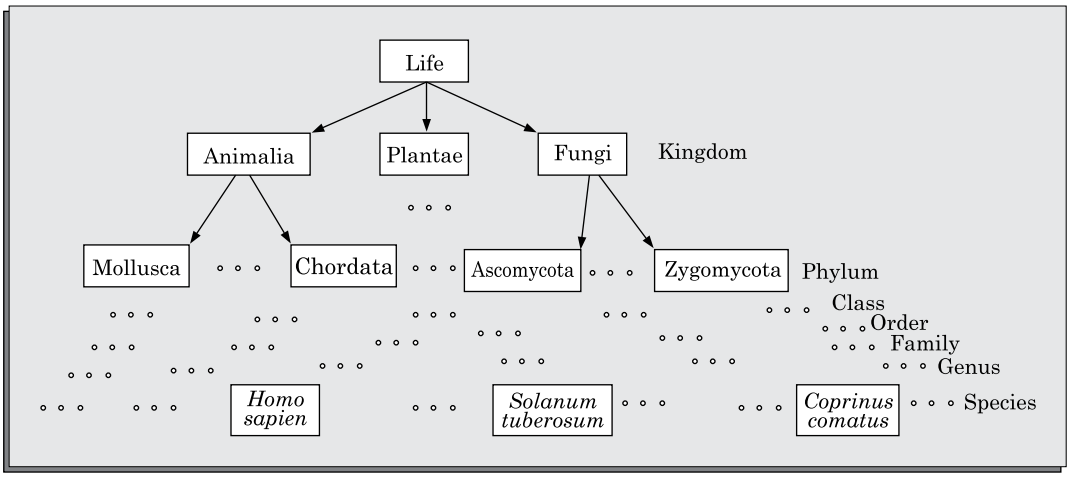


FIGURE 1.8 An abstraction hierarchy classifying living organisms.

A single level of abstraction can be sufficient for rather simple problems. However, more complex problems would need to be modelled as a hierarchy of abstractions. A schematic representation of an abstraction hierarchy has been shown in Figure 1.7(a). The most abstract representation would have only a few items and would be the easiest to understand. After one understands the simplest representation, one would try to understand the next level of abstraction where at most five or seven new information are added and so on until the lowest level is understood. By the time, one reaches the lowest level, he would have mastered the entire problem.

Decomposition

Decomposition is another important principle that is available in the repertoire of a software engineer to handle problem complexity. This principle is profusely made use by several software engineering techniques to contain the exponential growth of the perceived problem complexity. The decomposition principle is popularly known as the *divide and conquer* principle.

A popular way to demonstrate the decomposition principle is by trying to break a large bunch of sticks tied together and then breaking them individually. Figure 1.7(b) shows the decomposition of a large problem

The decomposition principle advocates decomposing the problem into many small independent parts. The small parts are then taken up one by one and solved separately. The idea is that each small part would be easy to grasp and understand and can be easily solved. The full problem is solved when all the parts are solved.

into many small parts. However, it is very important to understand that any arbitrary decomposition of a problem into small parts would not help. The different parts after decomposition should be more or less independent of each other. That is, to solve one part you should not have to refer and understand other parts. If to solve one part you would have to understand other parts, then this would boil down to understanding all the parts together. This would effectively reduce the problem to the original problem before decomposition (the case when all the sticks tied together). Therefore, it is not sufficient to just decompose the problem in any way, but the decomposition should be such that the different decomposed parts must be more or less independent of each other.

As an example of a use of the principle of decomposition, consider the following. You would understand a book better when the contents are decomposed (organised) into more or less independent chapters. That is, each chapter focuses on a separate topic, rather than when the book mixes up all topics together throughout all the pages. Similarly, each chapter should be decomposed into sections such that each section discusses a different issue. Each section should be decomposed into subsections and so on. If various subsections are nearly independent of each other, the subsections can be understood one by one rather than keeping on cross referencing to various subsections across the book to understand one.

Why study software engineering?

Let us examine the skills that you could acquire from a study of the software engineering principles. The following two are possibly the most important skill you could be acquiring after completing a study of software engineering:

- The skill to participate in development of large software. You can meaningfully participate in a team effort to develop a large software only after learning the systematic techniques that are being used in the industry.
- You would learn how to effectively handle complexity in a software development problem. In particular, you would learn how to apply the principles of abstraction and decomposition to handle complexity during various stages in software development such as specification, design, construction, and testing.

Besides the above two important skills, you would also be learning the techniques of software requirements specification user interface development, quality assurance, testing, project management, maintenance, etc.

As we had already mentioned, small programs can also be written without using software engineering principles. However even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity and at the same time enable you to produce better quality programs.

1.4 EMERGENCE OF SOFTWARE ENGINEERING

We have already pointed out that software engineering techniques have evolved over many years in the past. This evolution is the result of a series of innovations and accumulation of experience about writing good quality programs. Since these innovations and programming experiences are too numerous, let us briefly examine only a few of these innovations and programming experiences which have contributed to the development of the software engineering discipline.

1.4.1 Early Computer Programming

Early commercial computers were very slow and too elementary as compared to today's standards. Even simple processing tasks took considerable computation time on those computers. No wonder that programs at that time were very small in size and lacked sophistication. Those programs were usually written in assembly languages. Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code. Every programmer developed his own individualistic style of writing programs according to his intuition and used this style *ad hoc* while writing different programs. In simple words, programmers wrote programs without formulating any proper solution strategy, plan, or design a jump to the terminal and start coding immediately on hearing out the problem. They then went on fixing any problems that they observed until they had a program that worked reasonably well. We have already designated this style of programming as the *build and fix* (or the *exploratory programming*) style.

1.4.2 High-level Language Programming

Computers became faster with the introduction of the semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems. At this time, high-level languages such as FORTRAN, ALGOL, and COBOL were introduced. This considerably reduced the effort required to develop software and helped programmers to write larger programs (why?). Writing each high-level programming construct in effect enables the programmer to write several machine instructions. Also, the machine details (registers, flags, etc.) are abstracted from the programmer. However, programmers were still using the exploratory style of software development. Typical programs were limited to sizes of around a few thousands of lines of source code.

1.4.3 Control Flow-based Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of a program's *control flow structure*.

In order to help develop programs having good control flow structures, the *flow charting technique* was developed. Even today, the flow charting technique is being used to represent and design algorithms; though the popularity of the flow charting technique to represent and design programs has diminished due to the emergence of more advanced techniques.

A program's control flow structure indicates the sequence in which the program's instructions are executed.

Figure 1.9 illustrates two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.9 are drawn in Figure 1.10. Observe that the control flow structure of the program segment in Figure 1.10(b) is much simpler than that of Figure 1.10(a). By examining the code, it can be seen that Figure 1.10(a) is much harder to understand as compared to Figure 1.10(b).

```
1       if(customer_savings_balance>withdrawal_request) {
2   100:    issue_money=TRUE;
3           GOTO 110;
        }
4       else if(privileged_customer==TRUE)
5           GOTO 100;
6       else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8       GOTO 130;
9   120:    print(error);
10  130:    end–transaction();
```

(a) Unstructured program

```
1   if(privileged_customer||(customer_savings_balance>withdrawal_request)){
2           activate_cash_dispenser(withdrawal_request);
    }
3   else print(error);
4   end–transaction();
```

(b) Corresponding structured program

FIGURE 1.9 An example of (a) Unstructured program (b) Corresponding structured program.

This example corroborates the fact that if the flow chart representation is simple, then the corresponding code should be simple. You can draw the flow chart representations of several other problems to convince yourself that a program with complex flow chart representation is indeed more difficult to understand and maintain.

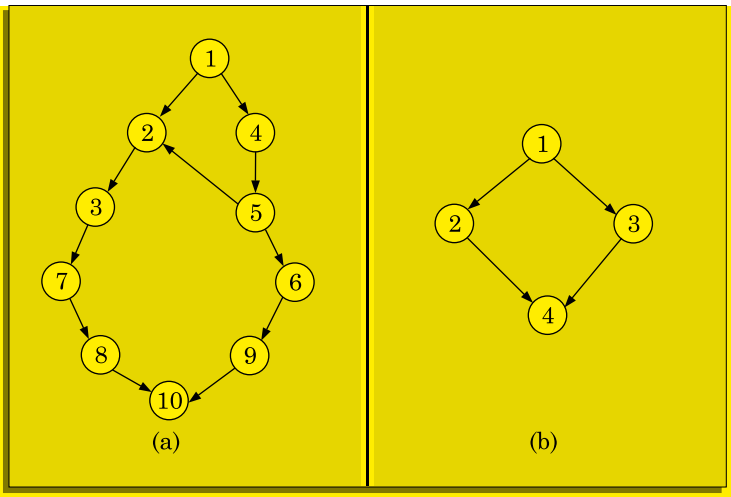


FIGURE 1.10 Control flow graphs of the programs of Figures 1.9(a) and (b).

Let us now try to understand why a program having good control flow structure would be easier to develop and understand. In other words, let us understand why a program with a complex flow chart representation is difficult to understand? The main reason behind this situation is that normally one understands a program by mentally tracing its execution sequence (i.e. statement sequences) to understand how the output is produced from the input values. That is, we can start from a statement producing an output, and trace back the statements in the program and understand how they produce the output by transforming the input data. Alternatively, we may start with the input data and check by running through the program how each statement processes (transforms) the input data until the output is produced. For example, for the program of Figure 1.10(a) you would have to

understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and 1-4-5-2-3-7-8-10. A program having a messy control flow (i.e. flow chart) structure, would have a large number of execution paths (see Figure 1.11). Consequently, it would become extremely difficult to determine all the execution paths, and tracing the execution sequence along all the paths trying to understand them can be nightmarish. It is therefore evident that a program having a messy flow chart representation would indeed be difficult to understand and debug.

A programmer trying to understand a program would have to mentally trace and understand the processing that take place along all the paths of the program making program understanding and debugging extremely complicated.

Are GO TO statements the culprits?

In a landmark paper, Dijkstra [1968] published his (now famous) article "GO TO Statements Considered Harmful". He pointed out that unbridled use of GO TO statements is the main culprit in making the control structure of a program messy. To understand his argument, examine Figure 1.11 which shows the flow chart representation of a program in which the programmer has used rather too many GO TO statements. GO TO statements alter the flow of control arbitrarily, resulting in too many paths. But, then why does use of too many GO TO statements makes a program hard to understand?

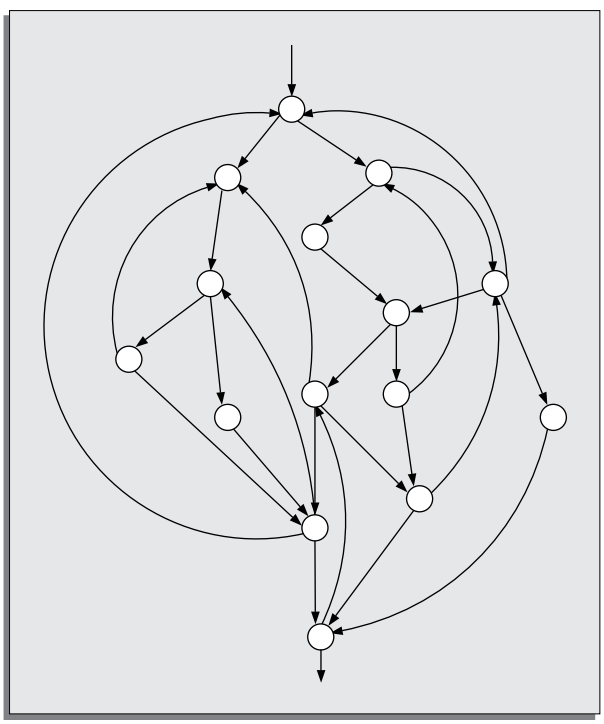


FIGURE 1.11 CFG of a program having too many GO TO statements.

Soon it became widely accepted that good programs should have very simple control structures. It is possible to distinguish good programs from bad programs by just visually

examining their flow chart representations. The use of flow charts to design good control flow structures of programs became wide spread.

Structured programming—a logical extension

The need to restrict the use of GO TO statements was recognised by everybody. However, many programmers were still using assembly languages. JUMP instructions are frequently used for program branching in assembly languages. Therefore, programmers with assembly language programming background considered the use of GO TO statements in programs inevitable. However, it was conclusively proved by Bohm and Jacopini [1966] that only three programming constructs—sequence, selection, and iteration—were sufficient to express any programming logic. This was an important result—it is considered important even today. An example of a sequence statement is an assignment statement of the form `a=b;`. Examples of selection and iteration statements are the `if-then-else` and the `do-while` statements respectively. Gradually, everyone accepted that it is indeed possible to solve any programming problem without using GO TO statements and that indiscriminate use of GO TO statements should be avoided. This formed the basis of the structured programming methodology.

Structured programs avoid unstructured control flows by restricting the use of GO TO statements. Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as `if-then-else`, `do-while`, etc. Thus, an important feature of structured programs is the design of good control structures. An example illustrating this key difference between structured and unstructured programs is shown in Figure 1.9. The program in Figure 1.9(a) makes use of too many GO TO statements, whereas the program in Figure 1.9(b) makes use of none. The control flow diagram of the program making use of GO TO statements is obviously much more complex as can be seen in Figure 1.10.

> A program is called *structured* when it uses only the sequence, selection, and iteration types of constructs and is modular.

Besides the control structure aspects, the term *structured program* is being used to denote a couple of other program features as well. A structured program should be modular. A modular program is one which is decomposed into a set of modules[1] such that the modules should have low interdependency among each other. We discuss the concept of modular programs in Chapter 5.

But, what are the main advantages of writing structured programs compared to the unstructured ones? Research experiences have shown that programmers commit less number of errors while using structured `if-then-else` and `do-while` statements than when using `test-and-branch` code constructs. Besides being less error-prone, structured programs are normally more readable, easier to maintain, and require less effort to develop compared to unstructured programs. The virtues of structured programming became widely accepted and the structured programming concepts are being used even today. However, violations to the structured programming feature is usually permitted in certain specific programming situations, such as exception handling, etc.

[1] In this text, we shall use the terms *module* and *module structure* to loosely mean the following—A *module* is a collection of procedures and data structures. The data structures in a module are accessible only to the procedures defined inside the module. A module forms an independently compilable unit and may be linked to other modules to form a complete application. The term *module structure* will be used to denote the way in which different modules invoke each other's procedures.

Very soon several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support structured programming. These programming languages facilitated writing modular programs and programs having good control structures. Therefore, messy control structure was no longer a big problem. So, the focus shifted from designing good control structures to designing good data structures for programs.

1.4.4 Data Structure-oriented Design

Computers became even more powerful with the advent of *integrated circuits* (ICs) in the early seventies. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. This often required writing in excess of several tens of thousands of lines of source code. The control flow-based program development techniques could not be used satisfactorily any more to write those programs, and more effective program development techniques were needed.

It was soon discovered that while developing a program, it is much more important to pay attention to the design of the important data structures of the program than to the design of its control structure. Design techniques based on this principle are called *data structure-oriented* design techniques.

Using data structure-oriented design techniques, first a program's data structures are designed. The code structure is designed based on the data structure.

In the next step, the program design is derived from the data structure. An example of a data structure-oriented design technique is the Jackson's Structured Programming (JSP) technique developed by Michael Jackson [1975]. In JSP methodology, a program's data structure is first designed using the notations for sequence, selection, and iteration. The JSP methodology provides an interesting technique to derive the program structure from its data structure representation. Several other data structure-based design techniques were also developed. Some of these techniques became very popular and were extensively used. Another technique that needs special mention is the Warnier-Orr Methodology [1977, 1981]. However, we will not discuss these techniques in this text because now-a-days these techniques are rarely used in the industry and have been replaced by the data flow-based and the object-oriented techniques.

1.4.5 Data Flow-oriented Design

As computers became still faster and more powerful with the introduction of *very large scale integrated* (VLSI) Circuits and some new architectural concepts, more complex and sophisticated software were needed to solve further challenging problems. Therefore, software developers looked out for more effective techniques for designing software and soon *data flow-oriented techniques* were proposed.

The data flow-oriented techniques advocate that the major data items handled by a system must be identified and the processing required on these data items to produce the desired outputs should be determined.

The functions (also called as *processes*) and the data items that are exchanged between the different functions are represented in a diagram known as a *data flow diagram* (DFD). The program structure can be designed from the DFD representation of the problem.

DFDs: A crucial program representation for procedural program design

DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, Figure 1.12 shows the data-flow representation of an automated car assembly plant. If you have never visited an automated car assembly plant, a brief description of an automated car assembly plant would be necessary. In an automated car assembly plant, there are several processing stations (also called *workstations*) which are located along side of a conveyor belt (also called an *assembly line*). Each workstation is specialised to do jobs such as fitting of wheels, fitting the engine, spray painting the car, etc. As the partially assembled program moves along the assembly line, different workstations perform their respective jobs on the partially assembled software. Each circle in the DFD model of Figure 1.12 represents a workstation (called a *process* or *bubble*). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be fitted from the corresponding stores (represented by two parallel horizontal lines), and as soon as the fitting work is complete passes on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in Figure 1.12 even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity. In Chapter 6, we shall study how to construct the DFD model of a software system. Once you develop the DFD model of a problem, data flow-oriented design techniques provide a rather straight forward methodology to transform the DFD representation of a problem into an appropriate software design. We shall study the data flow-based design techniques in Chapter 6.

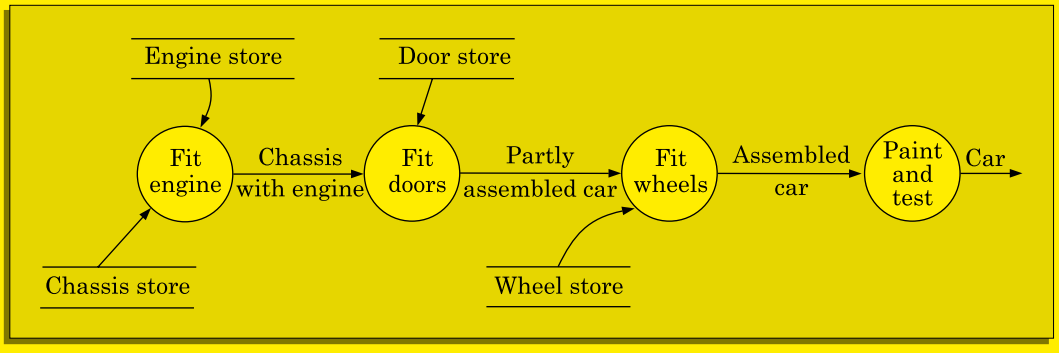


FIGURE 1.12 Data flow model of a car assembly plant.

1.4.6 Object-oriented Design

Data flow-oriented techniques evolved into *object-oriented design* (OOD) techniques in the late seventies. Object-oriented design technique is an intuitively appealing approach, where the natural objects (such as employees, pay-roll-register, etc.) relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a *data hiding* (also known as *data abstraction*) entity. Object-oriented techniques have gained wide spread acceptance because of their simplicity, the scope for code and design reuse, promise of lower development time,

lower development cost, more robust code, and easier maintenance. OOD techniques are discussed in Chapters 7 and 8.

1.4.7 What Next?

In this section, we have so far discussed how software design techniques have evolved since the early days of programming. We pictorially summarise this evolution of the software design techniques in Figure 1.13. It can be observed that in almost every passing decade, revolutionary ideas were put forward to design larger and more sophisticated programs, and at the same time the quality of the design solutions improved. But, what would the next improvement to the design techniques be? It is very difficult to speculate about the developments that may occur in the future. However, we have already seen that in the past, the design technique have evolved each time to meet the challenges faced in developing contemporary software. Therefore, the next development would most probably occur to help meet the challenges being faced by the modern software designers.

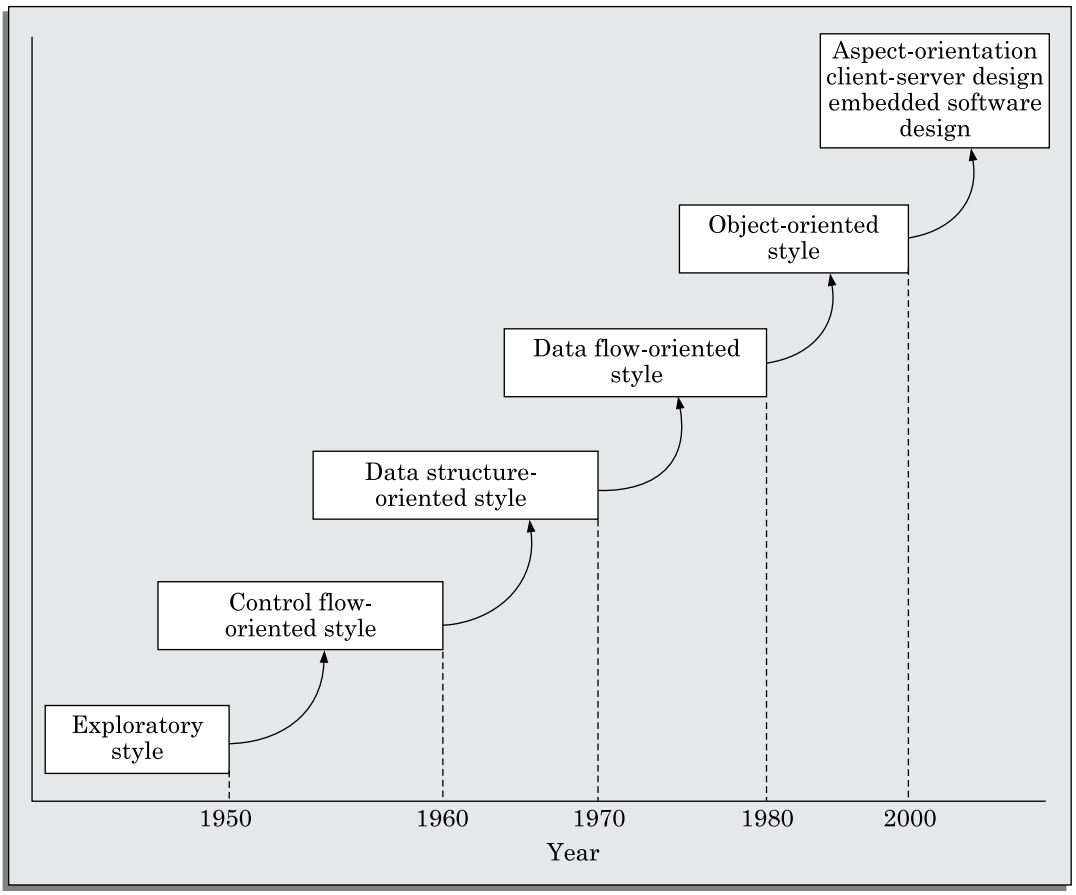


FIGURE 1.13 Evolution of software design techniques.

To get an indication of the techniques that are likely to emerge, let us first examine what are the current challenges in designing software. First, program sizes are further increasing as compared to what was being developed a decade back. Second, many of the present day software are required to work in a client-server environment through a web browser-based access (called *web-based software*). At the same time, embedded devices are experiencing an unprecedented growth and rapid customer acceptance in the last decade. It is there for necessary for developing applications for small hand held devices and embedded processors. We examine later in this text how aspect-oriented programming, client-server based design, and embedded software design techniques have emerged rapidly. In the current decade, service-orientation has emerged as a recent direction of software engineering due to the popularity of web-based applications and public clouds.

1.4.8 Other Developments

It can be seen that remarkable improvements to the prevalent software design technique occurred almost every passing decade. The improvements to the software design methodologies over the last five decades have indeed been remarkable. In addition to the advancements made to the software design techniques, several other new concepts and techniques for effective software development were also introduced. These new techniques include life cycle models, specification techniques, project management techniques, testing techniques, debugging techniques, quality assurance techniques, software measurement techniques, *computer aided software engineering* (CASE) tools, etc. The development of these techniques accelerated the growth of software engineering as a discipline. We shall discuss these techniques in the later chapters.

1.5 NOTABLE CHANGES IN SOFTWARE DEVELOPMENT PRACTICES

Before we discuss the details of various software engineering principles, it is worthwhile to examine the glaring differences that you would notice when you observe an exploratory style of software development and another development effort based on modern software engineering practices. The following noteworthy differences between these two software development approaches would be immediately observable.

- An important difference is that the exploratory software development style is based on *error correction (build and fix)* while the software engineering techniques are based on the principles of *error prevention*. Inherent in the software engineering principles is the realisation that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when mistakes are committed during development, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible (this topic is discussed in more detail in Section 2.3 of the next chapter). In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they are made.
- In the exploratory style, coding was considered synonymous with software development. For instance, this naive way of developing a software believed in

developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily. Exploratory programmers literally dive at the computer to get started with their programs even before they fully learn about the problem!!! It was recognised that exploratory programming not only turns out to be prohibitively costly for non-trivial problems, but also produces hard-to-maintain programs. Even minor modifications to such programs later can become nightmarish. In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which may demand much more effort than coding.

- A lot of attention is now being paid to requirements specification. Significant effort is being devoted to develop a clear and correct specification of the problem before any development activity starts. Unless the requirements specification is able to correctly capture the exact customer requirements, large number of rework would be necessary at a later stage. Such rework would result in higher cost of development and customer dissatisfaction.
- Now there is a distinct design phase where standard design techniques are employed to yield coherent and complete design models.
- Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is *phase containment of errors*, i.e. detect and correct errors as soon as possible. Phase containment of errors is an important software engineering principle. We will discuss this technique in Chapter 2.
- Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing, as test cases are being developed right from the requirements specification stage.
- There is better visibility of the software through various developmental activities.

  In the past, very little attention was being paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during software development. This has made fault diagnosis and maintenance far more smoother. We will see in Chapter 3 that in addition to facilitating product maintenance, increased visibility makes management of a software project easier.

  By visibility we mean production of good quality, consistent and peer reviewed documents at the end of every software development activity.

- Now, projects are being thoroughly planned. The primary objective of project planning is to ensure that the various development activities take place at the correct time and no activity is halted due to the want of some resource. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and automation tools for tasks such as configuration management, cost estimation, scheduling, etc., are being used for effective software project management.
- Several metrics (quantitative measurements) of the products and the product development activities are being collected to help in software project management and software quality assurance.

1.6 COMPUTER SYSTEMS ENGINEERING

In all the discussions so far, we assumed that the software being developed would run on some general-purpose hardware platform such as a desktop computer or a server. But, in several situations it may be necessary to develop special hardware on which the software would run.

Examples of such systems are numerous, and include a robot, a factory automation system, and a cell phone. In a cell phone, there is a special processor and other specialised devices such as a speaker and a microphone. It can run only the programs written specifically for it. Development of such systems entails development of both software and specific hardware that would run the software. Computer systems engineering addresses development of such systems requiring development of both software and specific hardware to run the software. Thus, systems engineering encompasses software engineering.

The general model of systems engineering is shown schematically in Figure 1.14. One of the important stages in systems engineering is the stage in which decision is made regarding the parts of the problems that are to be implemented in hardware and the ones that would be implemented in software. This has been represented by the box captioned hardware-software partitioning in Figure 1.14. While partitioning the functions between hardware and software, several trade-offs such as flexibility, cost, speed of operation, etc., need to be considered. The functionality implemented in hardware run faster. On the other hand, functionalities implemented in software is easier to extend. Further, it is difficult to implement complex functions in hardware. Also, functions implemented in hardware incur extra space, weight, manufacturing cost, and power overhead.

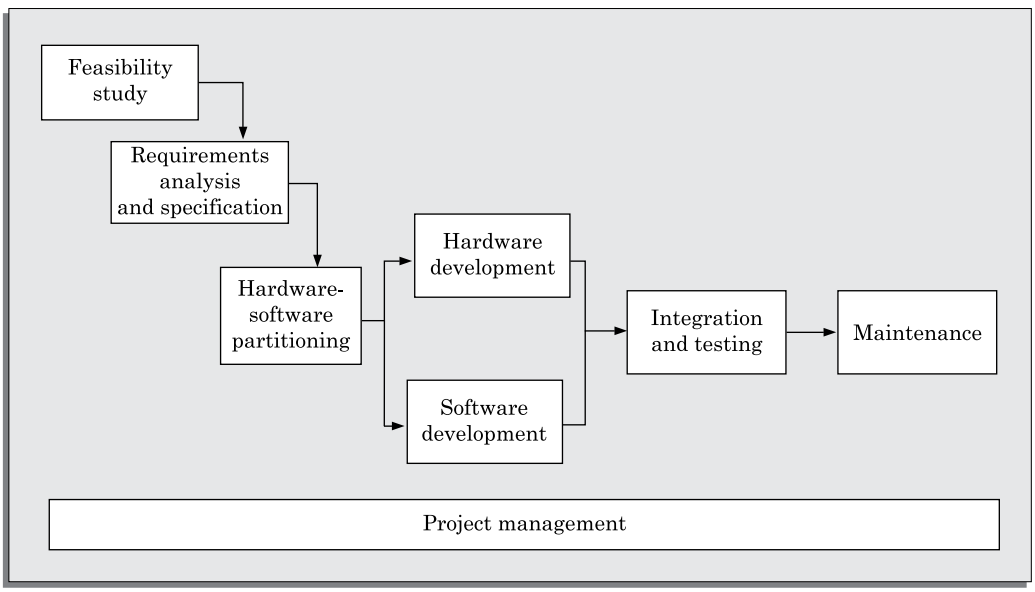


FIGURE 1.14 Computer systems engineering.

After the hardware-software partitioning stage, development of hardware and software are carried out concurrently (shown as concurrent branches in Figure 1.14). In system

engineering, testing the software during development becomes a tricky issue, the hardware on which the software would run and tested would still be under development—remember that the hardware and the software are being developed at the same time. To test the software during development, it usually becomes necessary to develop simulators that mimic the features of the hardware being developed. The software is tested using these simulators. Once both hardware and software development are complete, these are integrated and tested.

The project management activity is required through out the duration of system development as shown in Figure 1.14. In this text, we have confined our attention to software engineering only.

SUMMARY

- We first defined the scope of software engineering. We came up with two alternate but equivalent definitions:
  - The systematic collection of decades of programming experience together with the innovations made by researchers towards developing high quality software in a cost-effective manner.
  - The engineering approach to develop software.
- The exploratory (also called build and fix) style of program development is used by novice programmers. The exploratory style is characterized by quickly developing the program code and then modifying it successively till the program works. This approach turns out not only to be a very costly and inefficient way of developing software, but yields a product that is unreliable and difficult to maintain. Also, the exploratory style is very difficult to use when software is developed through team effort. A still larger handicap of the exploratory style of programming is that it breaks down when used to develop large programs.
- Unless one makes use of software engineering principles, the increase in effort and time with size of the program would be exponential—making it virtually impossible for a person to develop large programs.
- To handle complexity in a problem, all software engineering principles make extensive use of the following two techniques:
  - Abstraction (modelling), and
  - Decomposition (Divide and conquer).
- Software engineering techniques are essential for development of large software products where a group of engineers work in a team to develop the product. However, most of the principles of software engineering are useful even while developing small programs.
- A program is called structured, when it is decomposed into a set of modules and each module in turn is decomposed into functions. Additionally, structured programs avoid the use of GO TO statements and use only structured programming constructs.
- Computer systems engineering deals with the development of complete systems, necessitating integrated development of both software and hardware parts. Computer systems engineering encompasses software engineering.

- We shall delve into various software engineering principles starting from the next chapter. But, before that here is a word of caution. Those who have written large-sized programs, can better appreciate many of the principles of software engineering. Students with less or no programming experience would have to take our words for it and work harder with the topics. However, it is a fact that unless somebody has seen an elephant (read problems encountered during program development) at least once, any amount of describing and explaining would not result in the kind of understanding that somebody who has previously seen an elephant (developed a program) would get.

EXERCISES

MULTIPLE CHOICE QUESTIONS

For each of the following questions, only one of the options is correct. Choose the correct option:

1. Which of the following is not a symptom of the present software crisis?
(a) Software is expensive.
(b) It takes too long to build a software product.
(c) Software is delivered late.
(d) Software products are required to perform very complex tasks.

2. The goal of structured programming is which one of the following:
(a) To have well indented programs.
(b) To be able to infer the flow of control from the compiled code.
(c) To be able to infer the flow of control from the program text.
(d) To avoid the use of GO TO statements.

3. Unrestricted use of GO TO statements is normally avoided while writing a program, since:
(a) It increases the running time of programs.
(b) It increases memory requirements of programs.
(c) It results in larger executable code sizes.
(d) It makes debugging difficult.

4. Why is writing easily modifiable code important?
(a) Easily modifiable code results in quicker run time.
(b) Most real world programs require changes at some point of time or other.
(c) Most text editors make it mandatory to write modifiable code.
(d) Several people may be writing different parts of code at the same time.

5. Which one among the following is usually not considered a software service type of project?
(a) Software maintenance
(b) Software customization
(c) Outsourced software development
(d) Software product development

REVIEW QUESTIONS

1. What is the principal aim of the software engineering discipline? What does the discipline of software engineering discuss?
2. Why do you think systematic software development using the software engineering principle is any different than art or craft?
3. Distinguish between a program and a professionally developed software.
4. Distinguish among a program, a software product and a software service. Give one example of each. Discuss the difference of the characteristics of development projects for each of these.
5. What is a software product line? Give an example of a software product line. How is a software product line development any different from a software product development?
6. What are the main types of projects that are undertaken by software development companies? Give examples of these types of projects and point out the important characteristic differences between these types of projects.
7. Do you agree with the following statement—The focus of exploratory programming is error correction while the software engineering principles emphasise error prevention? Give the reasons behind your answer.
8. What difficulties would a software development company face, if it tries to use the exploratory (build and fix) program development style in its development projects? Explain your answer.
9. What are the symptoms of the present software crisis? What factors have contributed to the making of the present software crisis? What are the possible solutions to the present software crisis?
10. Explain why the effort, time, and cost required to develop a program using the build and fix style increase exponentially with the size of the program? How do software engineering principles help tackle this rapid rise in development time and cost?
11. Distinguish between software products and services. Give examples of each.
12. What are the different types of projects that are being undertaken by software development houses? Which of these type of projects is the forte of Indian software development organisations? Identify any possible reasons as to why the other has not been focused by the Indian software development organisations.
13. Name the basic techniques used by the software engineering techniques to handle complexity in a problem.
14. What do you understand by the exploratory (also known as the build and fix) style of software development? Graphically depict the activities that a programmer typically carries out while developing a programming solution using the exploratory style. In your diagram also show the order in which the activities are carried out. What are the shortcomings of this style of program development?

15. List the major differences between the exploratory and modern software development practices.
16. What is the difference between the actual complexity of solving a problem and its perceived complexity? What causes the difference between the two to arise?
17. What do you understand by the term perceived complexity of a problem? How is it different from computational complexity? How can the perceived complexity of a problem be reduced?
18. Why is the number 7 considered as a magic number in software engineering? How is it useful software engineering?
19. What do you understand by the principles of abstraction and decomposition? Why are these two principles considered important in software engineering? Explain the problems that these two principles target to solve? Support your answer using suitable examples.
20. What do you understand by control flow structure of a program? Why is it difficult to understand a program having a messy control flow structure? How can a good control flow structure for a program be designed?
21. What is a flow chart? How is the flow charting technique useful during software development?
22. What do you understand by visibility of design and code? How does increased visibility help in systematic software development? (We shall revisit this question in Chapter 3)
23. What do you understand by the term—structured programming? How do modern programming languages such as PASCAL and C facilitate writing structured programs? What are the advantages of writing structured programs vis-a-vis unstructured programs?
24. What is a high-level programming language? Why does a programmer using a high-level programming language have a higher productivity as compared to when using machine language for application development?
25. What are the three basic types of program constructs necessary to develop the program for any given problem? Give examples of these three constructs from any high-level language you know.
26. What do you understand by a program module? What are the important characteristics of a program module?
27. Explain how do the use of software engineering principles help to develop software products cost-effectively and timely. Elaborate your answer by using suitable examples.
28. What is the basic difference between a control flow-oriented and a data flow-oriented design technique? Can you think of any reason as to why a data flow-oriented design technique is likely to produce better designs than a control flow-oriented design technique? (We shall revisit this question while discussing the design techniques in Chapter 6.)
29. Name the two fundamental principles that are used extensively in software engineering to tackle the complexity in developing large programs? Explain these

two principles. By using suitable examples explain how these two principles help tackle the complexity associated with developing large programs.

30. What does the *control flow graph* (CFG) of a program represent? Draw the CFG of the following program:

```
main(){
int  y=1;
    if(y<0)
        if(y>0)   y=3;
        else  y=0;
    printf("%d\n",y);
}
```

31. Discuss the possible reasons behind supersession of the data structure-oriented design methods by the control flow-oriented design methods.

32. What is a data structure-oriented software design methodology? How is it different from the data flow-oriented design methodology?

33. Discuss the major advantages of the *object-oriented design* (OOD) methodologies over the data flow-oriented design methodologies.

34. Explain how the software design techniques have evolved in the past. How do you think shall the software design techniques evolve in the near future?

35. What is computer systems engineering? How is it different from software engineering? Give examples of some types of product development projects for which systems engineering is appropriate.

36. What do you mean by software service? Explain the important differences between the characteristics of a software service development project and a software product development project.

37. What do you understand by the principles of abstraction and decomposition? Briefly explain these two techniques. Explain how these two techniques help to tackle the complexity of a programming problem by using suitable examples.

38. What do you understand by software product line? Give an example of a software product line.

39. Briefly explain why the early programmers can be considered to be similar to artists, the later programmers to be more like craftsmen, and the modern programmers to be engineers.