UNIT-I

1. INTRODUCTION

2. Software Life Cycle Models

UNIT-I

Chapter 1

INTRODUCTION

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".

EXAMPLE:-

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.

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.

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.

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. the exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

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.

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.

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 organised into a body of knowledge that forms the discipline of software engineering.

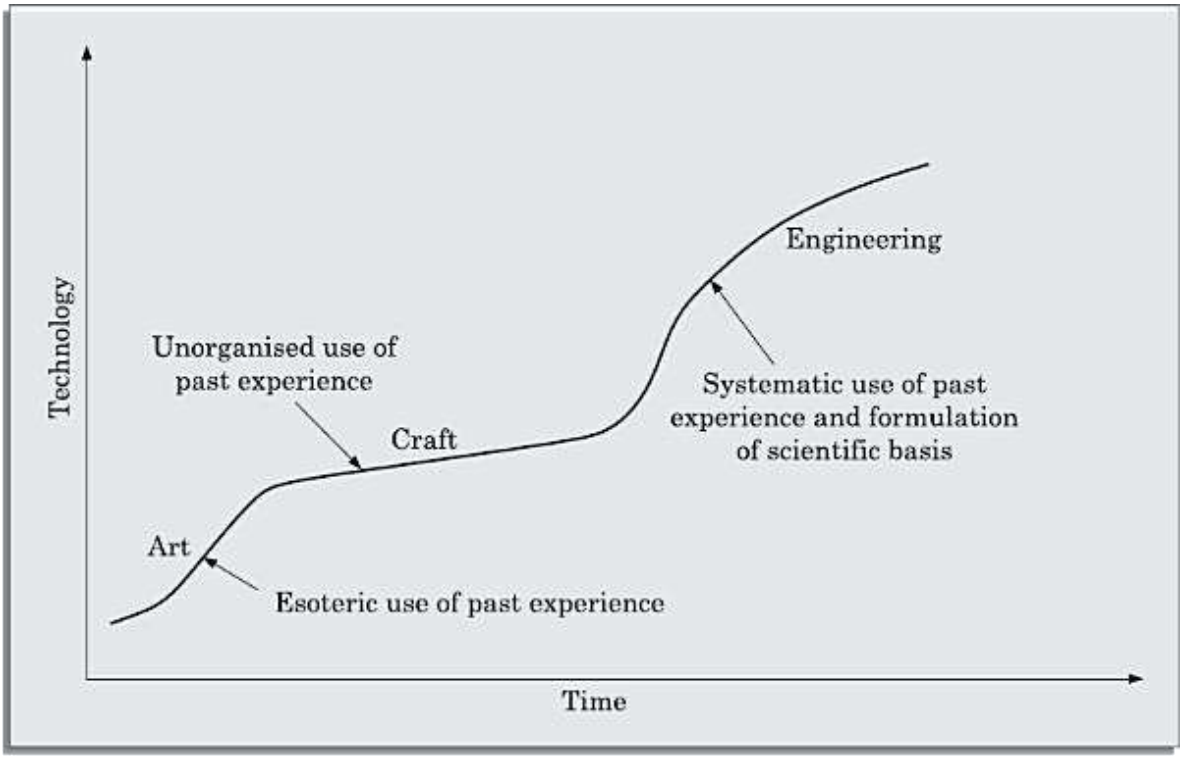


Figure 1.1: Evolution of technology with time.

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. 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.

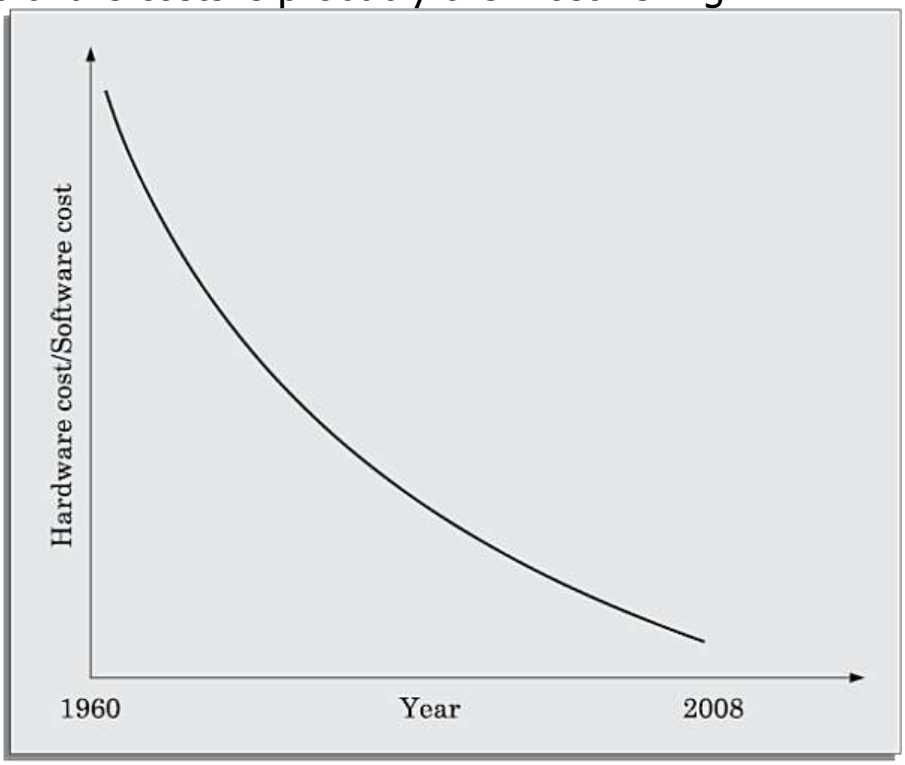


Figure 1.2: Relative changes of hardware and software costs over time.

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!!!

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 first programming assignment.

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*.

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.

1.2.1 Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects. These software development projects concern the development of either software product or some software service. In the following subsections, we distinguish between these two types of software development projects.

Software products

We all know of a variety of software such as Microsoft's Windows and the Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. 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.

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.

Software services

A software service usually involves either development of a *customised software* or development of some specific part of a software in an outsourced mode. A *customised software* is developed according to the specification drawn up by one or at most a few customers. These need to be developed in a short time frame (typically a couple of months), and at the same time the development cost must be low. Usually, a developing company develops customised software by tailoring some of its existing software.

For example, when an academic institution wishes to have a software that would automate its important activities such as student registration, grading, and fee collection; companies would normally develop such a software as a customised product.

Another type of software service is *outsourced software*. 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 if it determines 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.

1.2.2 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.

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.

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 1.3.

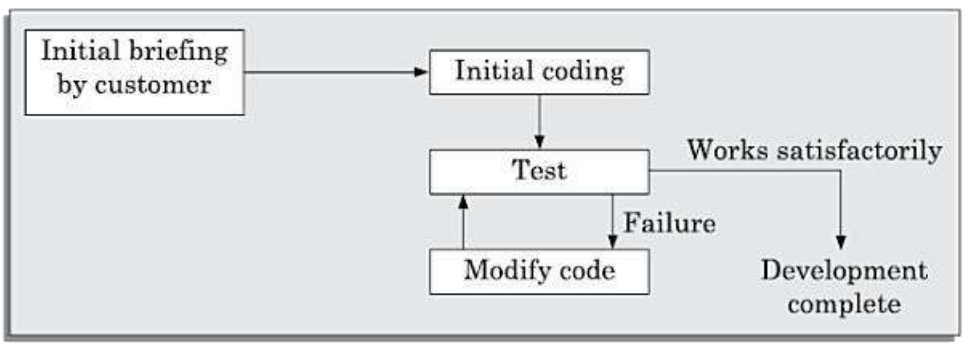


Figure 1.3: Exploratory program development.

Observe that coding starts after an initial customer briefing 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 1.4. 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. 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.

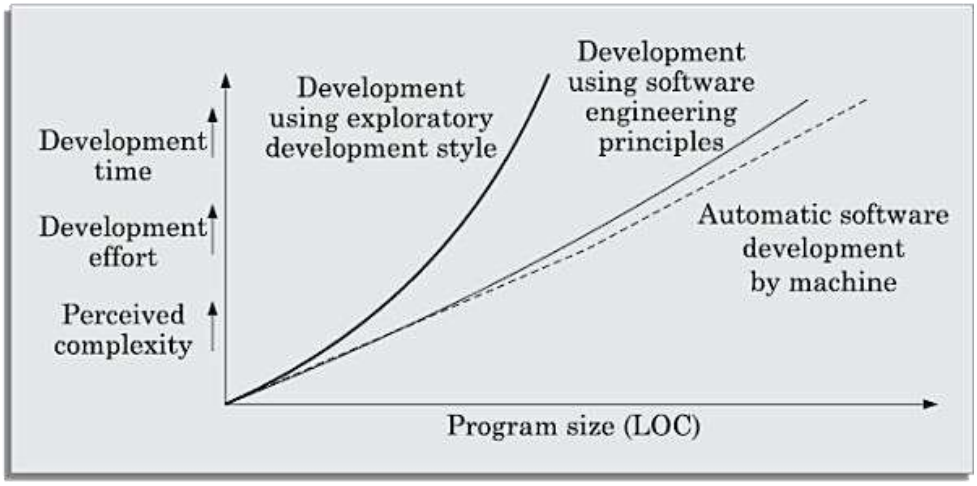


Figure 1.4: Increase in development time and effort with problem size.

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?

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.

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. 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.

Psychologists say that the human memory can be thought to consist of two distinct parts[Miller 56]: 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 1.5. In Figure 1.5, 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.

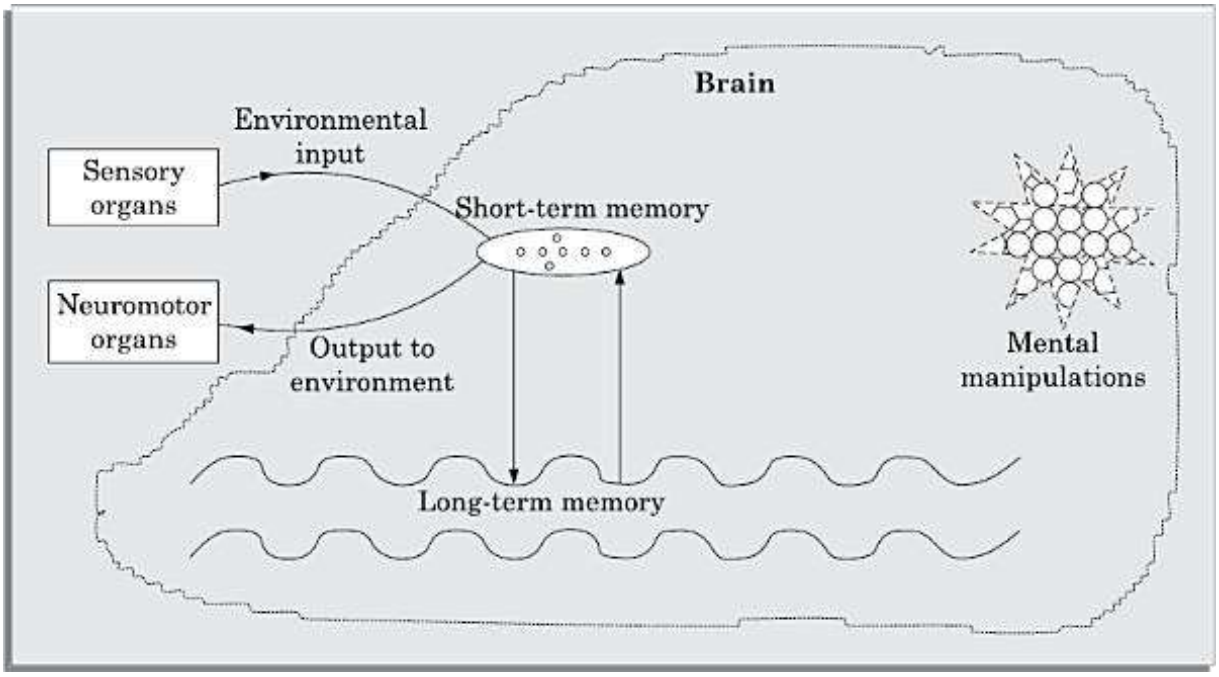


Figure 1.5: Human cognition mechanism model.

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 1.5, 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. As you can notice, this model is very similar to the organisation of a computer in terms of cache, main memory, and processor.

Long-term memory: 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 y o u 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 e a ch one less than the corresponding digits of the previous number.

> 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.

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 1.4. Please note that the situation depicted in Figure 1.4 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, a s 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.

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.

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

In the following subsections, with the help of Figure 1.6(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.

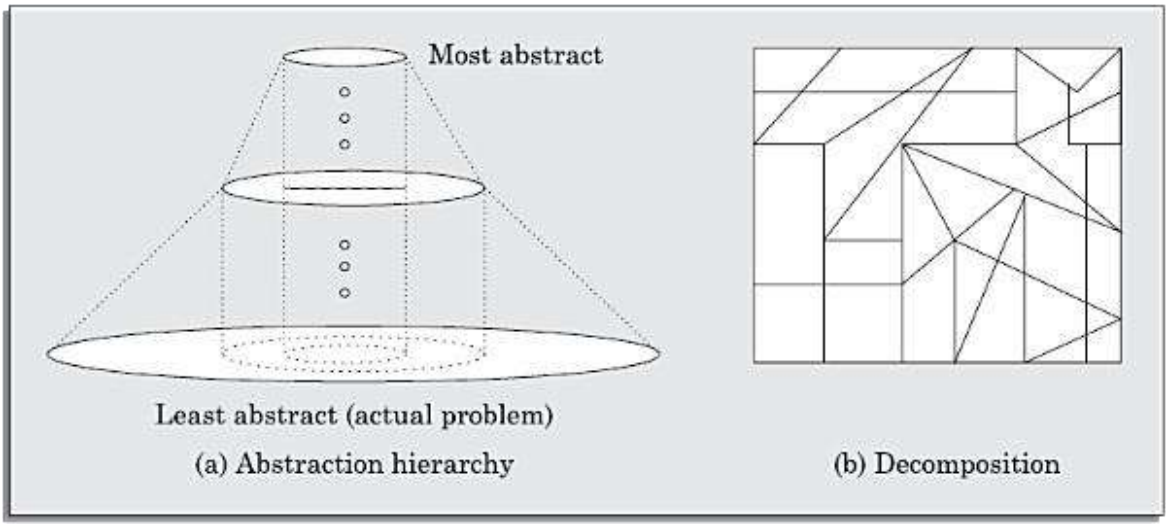


Figure 1.6: Schematic representation.

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*).

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

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.

- 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 1.7. 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.

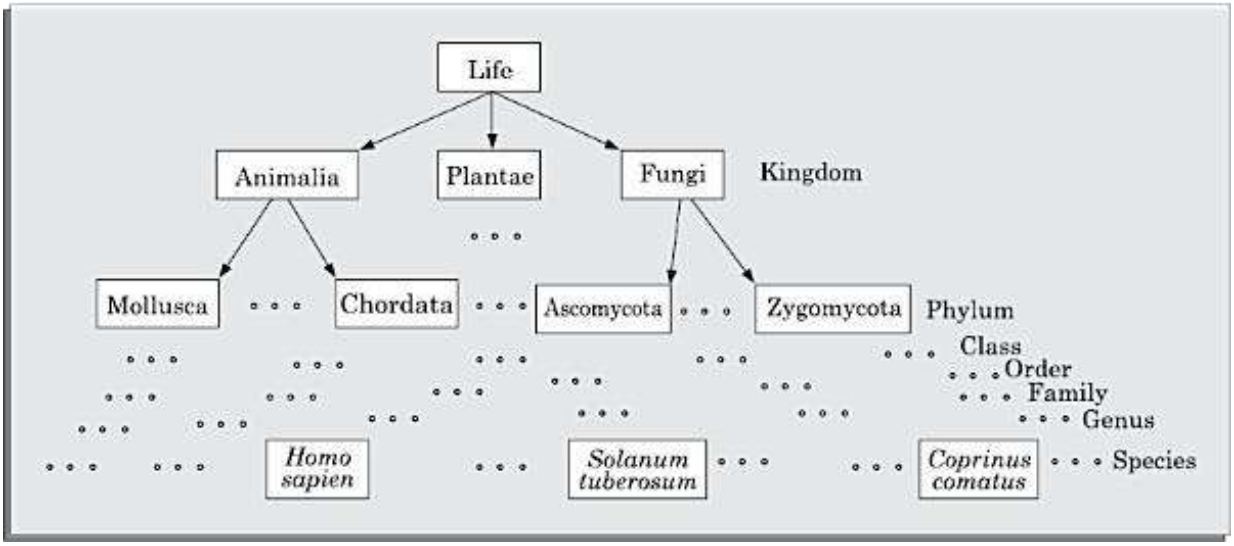


Figure 1.7: An abstraction hierarchy classifying living organisms.

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 1.6(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.

> 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.

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 1.6(b) shows the decomposition o f a large problem into many small parts.

As an example o f 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.

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 t o 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*.

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

In order to help develop programs having good control flow structures, the *flow charting technique* was developed. E v e n today, t h e flow charting technique is being used to represent and design algorithms; though the popularity of flow charting represent and design programs has want to a great extent due to the emergence of more advanced techniques.

1.4.4 Data Structure-oriented Design

Computers became even more powerful with the advent o f *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. which 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.

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 *d a t a flow-oriented techniques* were proposed.

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

For example, Figure 1.11 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 a s 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 DFDmodel of Figure 1.11 represents a workstation (called a *process* o r *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 1.11 even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity.

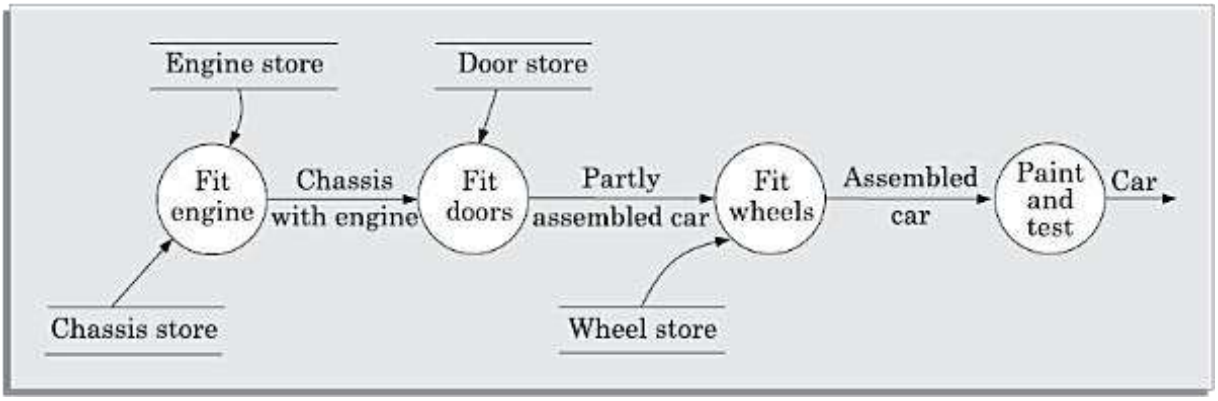


Figure 1.11: Data flow model of a car assembly plant.

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 a r e 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.

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. However, we have already seen that in the past, the design techniques 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. 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.

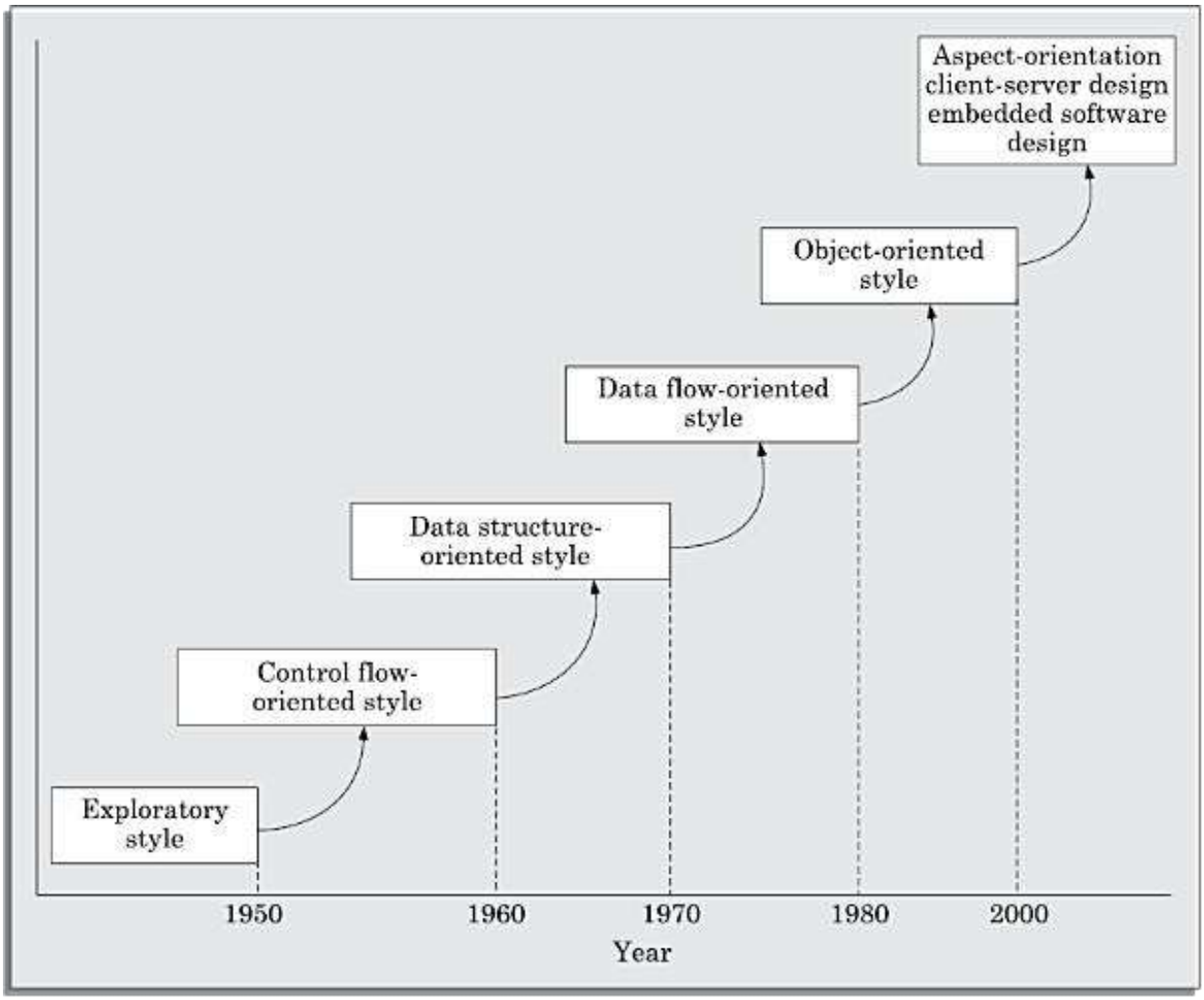


Figure 1.12: Evolution of software design techniques.

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 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.
- 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. I t 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 1.12. One of the important stages in systems engineering i s 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 1.13. 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 had, 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.

After the hardware-software partitioning stage, development of hardware and software are carried out concurrently (shown as concurrent branches in Figure 1.13). 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 1.13. In this text, we have confined our attention to software engineering only.

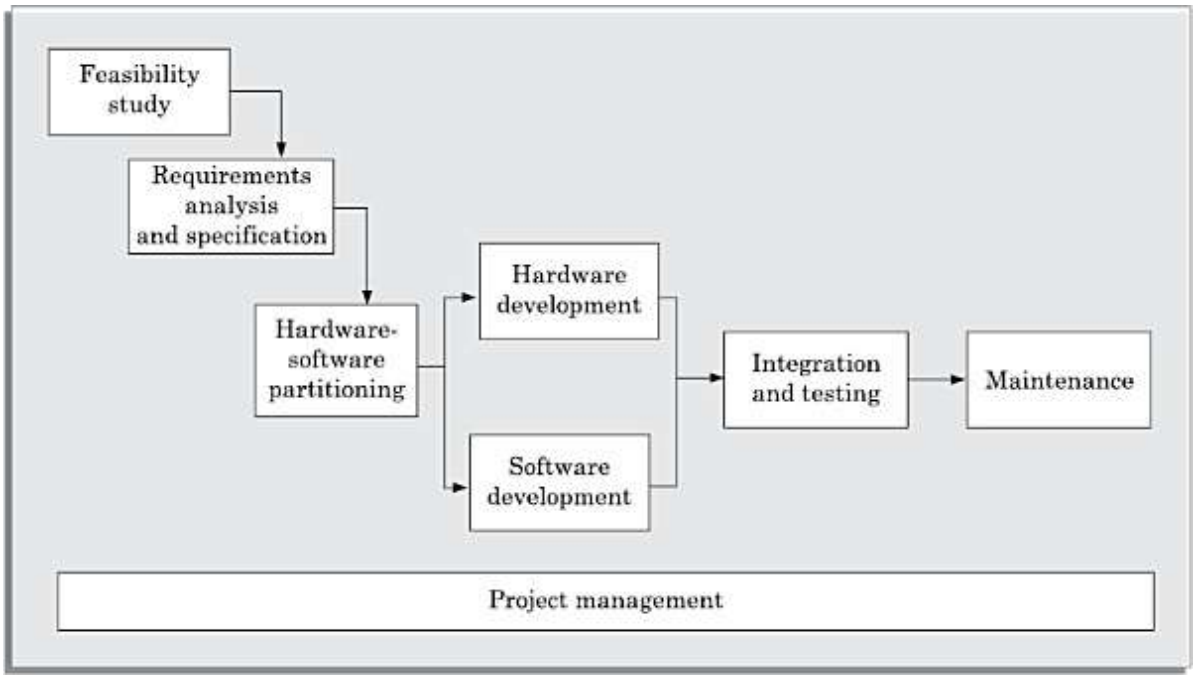


Figure 1.13: Computer systems engineering.

Chapter 2

SOFTWARE LIFE CYCLE MODELS

2.1 WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to certain specific software development situations to realise all other software life cycle models. For this reason, after discussing the classical and iterative waterfall models, we discuss its various extensions.

2.1.1 Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project. One might wonder if this model is hard to use in practical development projects, then why study it at all? The reason is that all other life cycle models can be thought of as being extensions of the classical waterfall model.

Therefore, it makes sense to first understand the classical waterfall model, in order to be able to develop a proper understanding of other life cycle models. Besides, we shall see later in this text that this model though not used for software development; is implicitly used while documenting software.

The classical waterfall model divides the life cycle into a set of phases as shown in Figure 2.1. It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the model.

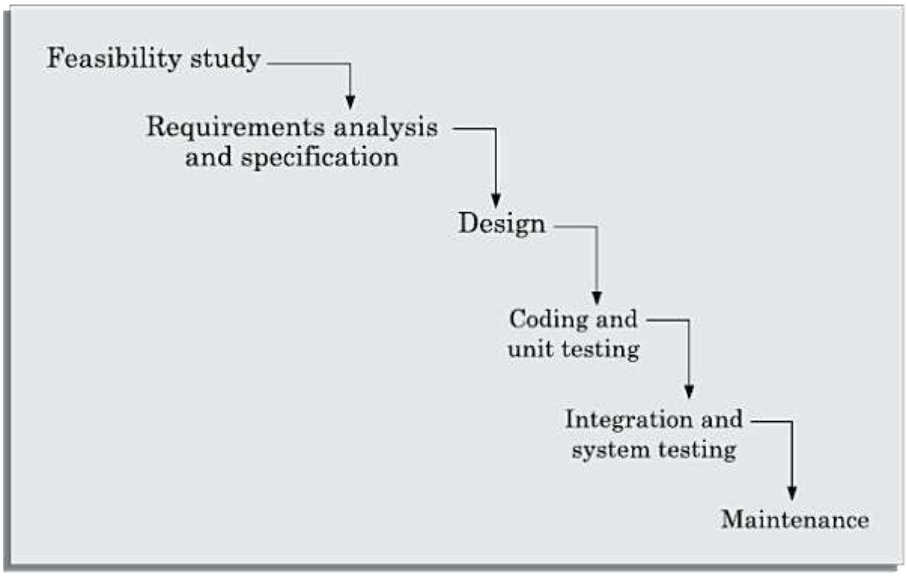


Figure 2.1: Classical waterfall model.

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in Figure 2.1. As shown in Figure 2.1, the different phases are—feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the *development phases*.

A software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signalling the commencement of the operation phase. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken. Therefore, the last phase is also known as the *maintenance phase* of the life cycle.

In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team. The relative amounts of effort spent on different phases for a typical software has been shown in Figure 2.2. Observe from Figure 2.2 that among all the life cycle phases, the maintenance phase normally requires the maximum effort. On the average, about 60 per cent of the total effort put in by the development team in the entire life cycle is spent on the maintenance activities alone.

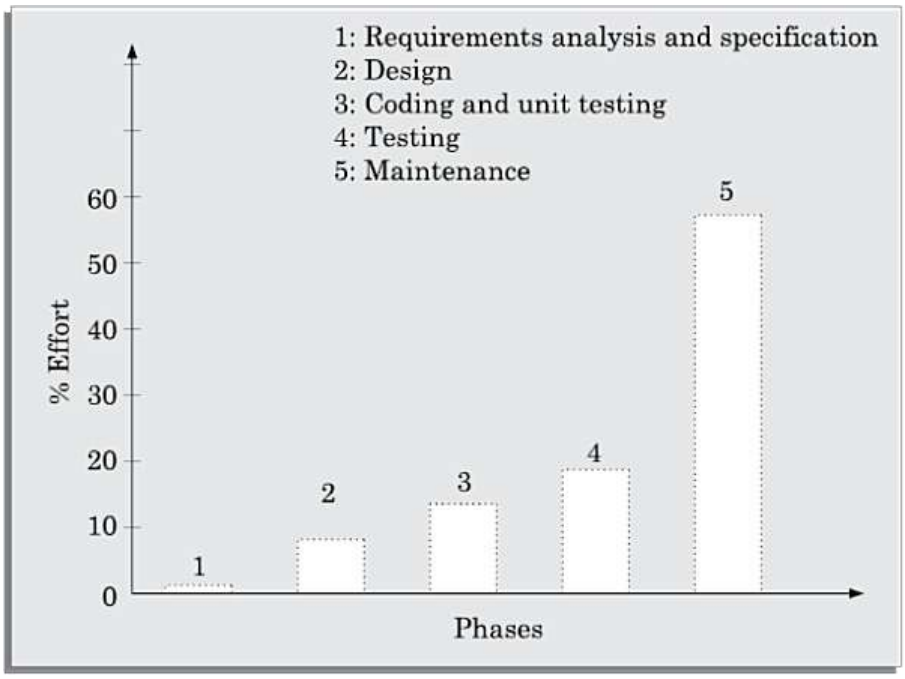


Figure 2.2: Relative effort distribution among different phases of a typical product.

However, among the development phases, the integration and system testing phase requires the maximum effort in a typical development project. In the following subsection, we briefly describe the activities that are carried out in the different phases of the classical waterfall model.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software. The feasibility study involves carrying out several activities such as collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analysed to perform at the following:

Development of an overall understanding of the problem: It is necessary to first develop an overall understanding of what the customer requires to be developed. For this, only the the important requirements of the customer need to be understood and the details of various requirements such as the screen layouts required in the *graphical user interface* (GUI), specific formulas or algorithms required for producing the required results, and the databases schema to be used are ignored.

Formulation of the various possible strategies for solving the problem: In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.

Evaluation of the different solution strategies: The different identified solution schemes are analysed to evaluate their benefits and shortcomings. Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required. The different solutions are compared based on the estimations that have been worked out. Once the best solution is identified, all activities in the later phases are carried out as per this solution. At this stage, it may also be determined that none of the solutions is feasible due to high cost, resource constraints, or some technical reasons. This scenario would, of course, require the project to be abandoned.

We can summarise the outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not, at this stage very high-level decisions regarding the solution strategy is defined. Therefore, feasibility study is a very crucial stage in software development. The following is a case study of the feasibility study undertaken by an organisation. It is intended to give a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification. In the following subsections, we give an overview of these two activities:

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that a n *inconsistent* requirement is one in which

some part of the requirement contradicts with some other part. On the other hand, a n *incomplete* requirement is one in which some parts of the actual requirements have been omitted.

- **Requirements specification:** After the requirement gathering and analysis activities are complete, the identified requirements are documented. This is called a *software requirements specification* (SRS) document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Therefore, understandability of the SRS document is an important issue. The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer. The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it. In Chapter 4, we examine the requirements analysis activity and various issues involved in developing a good SRS document in more detail.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the *software architecture* is derived from the SRS document. Two distinctly different design approaches are popularly being used at present—the procedural and object-oriented design approaches.

- **Procedural design approach:** The traditional design approach is in use in many software development projects at the present time. This traditional design technique is based on the data flow-oriented design approach. It consists of two important activities; first *structured analysis* of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a *structured design* step where the results of structured analysis are transformed into the software design.

During structured analysis, the functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analysed and represented diagrammatically in the form of DFDs. The DFD technique is discussed in Chapter 6. Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities—architectural design (also called *high-level design*) and detailed design (also called *Low-level design*). High-level design involves decomposing the system i nt o modules, and representing the interfaces and the invocation relationships among the modules..

- **Object-oriented design approach:** In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design. The OOD approach is credited to have several benefits such as lower development time and effort, and better maintainability of the software.

Coding and unit testing

The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly. The coding phase is also sometimes called t h e *implementation phase*, since the design is implemented into a workable solution in this phase. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested. The main objective of unit testing is to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases.

Integration and system testing

Integration of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Various modules making up a software are almost never integrated in one shot (can you guess the reason for this?). Integration of various modules are normally carried out

incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained. System testing is carried out on this fully working system.

Maintenance

The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself. Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60. Maintenance is required in the following three types of situations:

- **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
- **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Shortcomings of the classical waterfall model

The classical waterfall model is a very simple and intuitive model. However, it suffers from several shortcomings. Let us identify some of the important shortcomings of the classical waterfall model:

No feedback paths: In classical waterfall model, the evolution of a software from one phase to the next is analogous to a waterfall. Just as water in a waterfall after having flowed down cannot flow back, once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework. This requires that all activities during a phase are flawlessly carried out.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project. There is much emphasis on creating an

unambiguous and complete set of requirements. But, it is hard to achieve this even in ideal project scenarios. The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods.

Is the classical waterfall model useful at all?

The rationale behind preparation of documents based on the classical waterfall model can be explained using Hoare's metaphor of mathematical theorem [1994] proving—A mathematician presents a proof as a single chain of deductions, even though the proof might have come from a convoluted set of partial attempts, blind alleys and backtracks. Imagine how difficult it would be to understand, if a mathematician presents a proof by retaining all the backtracking, mistake corrections, and solution refinements he made while working out the proof.

2.1.2 Iterative Waterfall Model

We had pointed out in the previous section that in a practical software development project, the classical waterfall model is hard to use. We had branded the classical waterfall model as an idealistic model. In this context, the iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

> The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later

phase. For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to b e reflected in the design documents and all other subsequent documents. Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team having accepted to take up a project, does not give up the project easily due to legal and moral reasons.

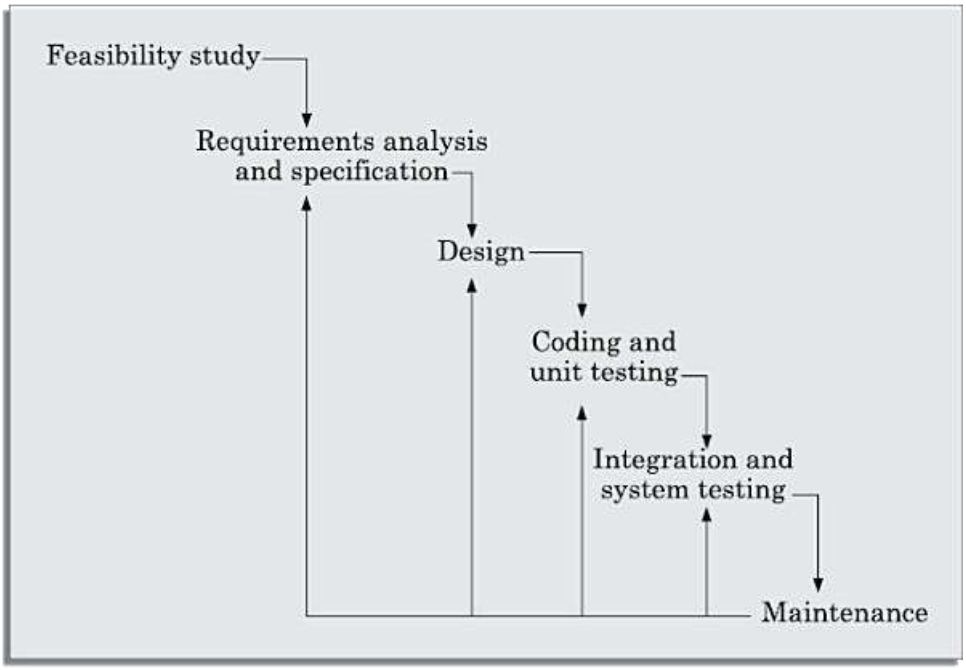


Figure 2.3: Iterative waterfa**l** model.

Almost every life cycle model that we discuss are iterative in nature, except the classical waterfall model and the V-model—which are sequential in nature. In a sequential model, once a phase is complete, no work product of that phase are changed later.

Phase containment of errors

No matter how careful a programmer may be, he might end up committing some mistake or other while carrying out a life cycle activity. These mistakes result in errors (also called *faults* o r *bugs*) in the work product. It is advantageous to detect these errors in the same phase in which they take place, since early detection of bugs reduces the effort and time required for correcting those. For example, if a design problem i s detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the testing activities, thereby incurring higher cost. It may not always be possible to detect all the errors in the same

phase in which they are made. Nevertheless, the errors should be detected as early as possible.

For achieving phase containment of errors, how can the developers detect almost all error that they commit in the same phase? After all, the end product of many phases are text or graphical documents, e.g. SRS document, design document, test plan document, etc. A popular technique is to rigorously review the documents produced at the end of a phase.

Phase overlap

Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next (see Figure 2.3), in practice the activities of different phases overlap (as shown in Figure 2.4) due to two main reasons:

- In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. These subsequently detected errors cause the activities of some already completed phases to be reworked. If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.
- An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a *blocking state*. Thus the developers who complete early would idle while waiting for their team mates to complete their assigned work. Clearly this is a cause for wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap. That is, once a developer completes his work assignment for a phase, proceeds to start the work for the next phase, without waiting for all his team members to complete their respective work allocations.

Considering these situations, the effort distribution for different phases with

time would be as shown in Figure 2.4.

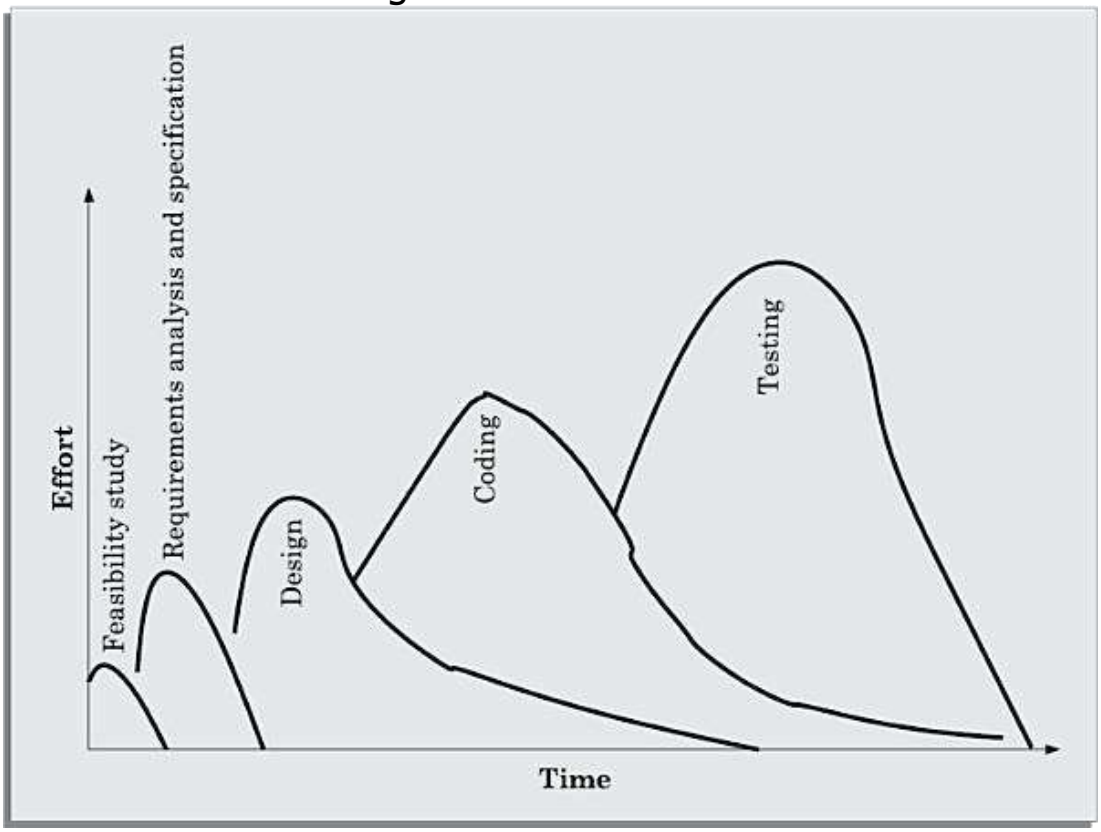


Figure 2.4: Distribution of effort for various phases in the iterative waterfall model.

Shortcomings of the iterative waterfall model

Some of the glaring shortcomings of the waterfall model when used in the present-day software development projects are as following:

Difficult to accommodate change requests: A major problem with the waterfall model is that the requirements need to be frozen before the development starts. Based on the frozen requirements, detailed plans are made for the activities to be carried out during the design, coding, and testing phases. Since activities are planned for the entire duration, substantial effort and resources are invested in the activities as developing the complete requirements specification, design for the complete functionality and so on. Therefore, accommodating even small change requests after the development activities are underway not only requires overhauling the plan, but also the artifacts that have already been developed.

The basic assumption made in the iterative waterfall model that methodical requirements gathering and analysis alone would comprehensively and correctly identify all the requirements by the end of the requirements phase is flawed.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur. This is problematic because the complete application may take several months or years to be completed and delivered to the customer. By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.

Phase overlap not supported: For most real life projects, i t becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*. The waterfall model is usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure 2.4.

Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.

Heavy weight: The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.

No support for risk handling and code reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of projects usually involve significant reuse.

2.1.3 V-Model

A popular development process model, V-model is a variant of the waterfall model. As is the case with the waterfall model, this model gets its name from its visual appearance (see Figure 2.5). In this model verification and validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce. This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability.

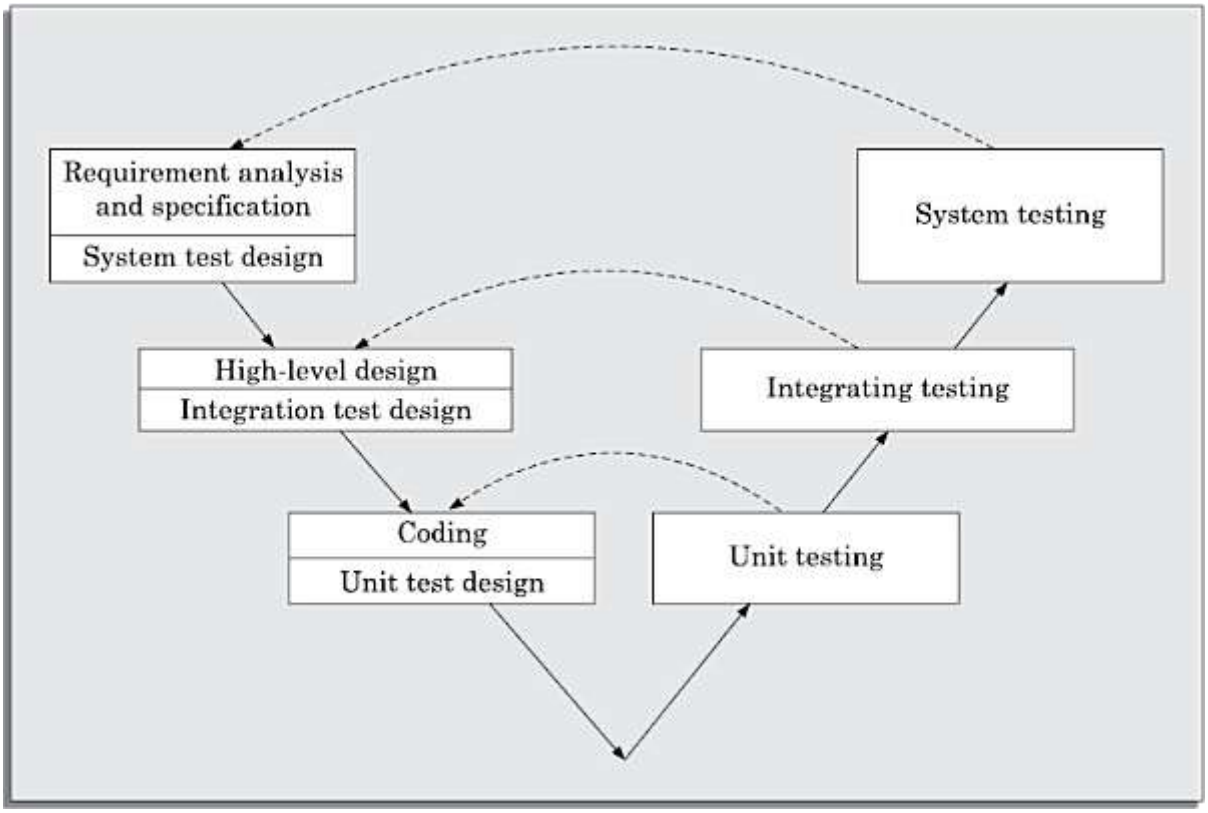


Figure 2.5: V-model.

As shown in Figure 2.5, there are two main phases—development and validation phases. The left half of the model comprises the development phases and the right half comprises the validation phases.

- In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which have been shown by dotted arcs in Figure 2.5.

- In the validation phase, testing is carried out in three steps—unit, integration, and system testing. The purpose of these three different steps of testing during the validation phase is to detect defects that arise in the corresponding phases of software development—requirements analysis and specification, design, and coding respectively.

V-model *versus* waterfall model

We have already pointed out that the V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. As already mentioned, in contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle. As shown in Figure 2.5, during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

Advantages of V-model

The important advantages of the V-model over the iterative waterfall model are as following:

- In the V-model, much o f the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.
- Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.
- The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.
- In the V-model, the test team is associated with the project from the

beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle,since no testing activities are carried out before the start of the implementation and testing phase.

Disadvantages of V-model

Being a derivative of the classical waterfall model, this model inherits most of the weaknesses of the waterfall model.

2.1.4 Prototyping Model

The prototype model is also a popular life cycle model. The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working *prototype* of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software.

A prototype can be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. Normally the term *rapid prototyping* is used when software tools are used for prototype construction. For example, tools based on *fourth generation languages* (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

The prototyping model is advantageous to use for specific types of projects. In the following, we identify three types of projects for which the prototyping model can be followed to advantage:

- It is advantageous to use the prototyping model for development of the *graphical user interface* (GUI) part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer. This is a valuable mechanism for gaining better

understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the *graphical user interface* (GUI) part of a system. For the user, it becomes much easier to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a hypothetical user interface.

- The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development.
- An important reason for developing a prototype is that it is impossible to "get it right" the first time. As advocated by Brooks [1975], one must plan to throw away the software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimised and efficient software is required.

From the above discussions, we can conclude the following:

Life cycle activities of prototyping model

The prototyping model of software development is graphically shown in Figure 2.6. As shown in Figure 2.6, software is developed through two major activities—prototype construction and iterative waterfall-based software development.

Prototype development: Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype that has been approved by

the customer serves as an animated requirements specification.

T h e code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.

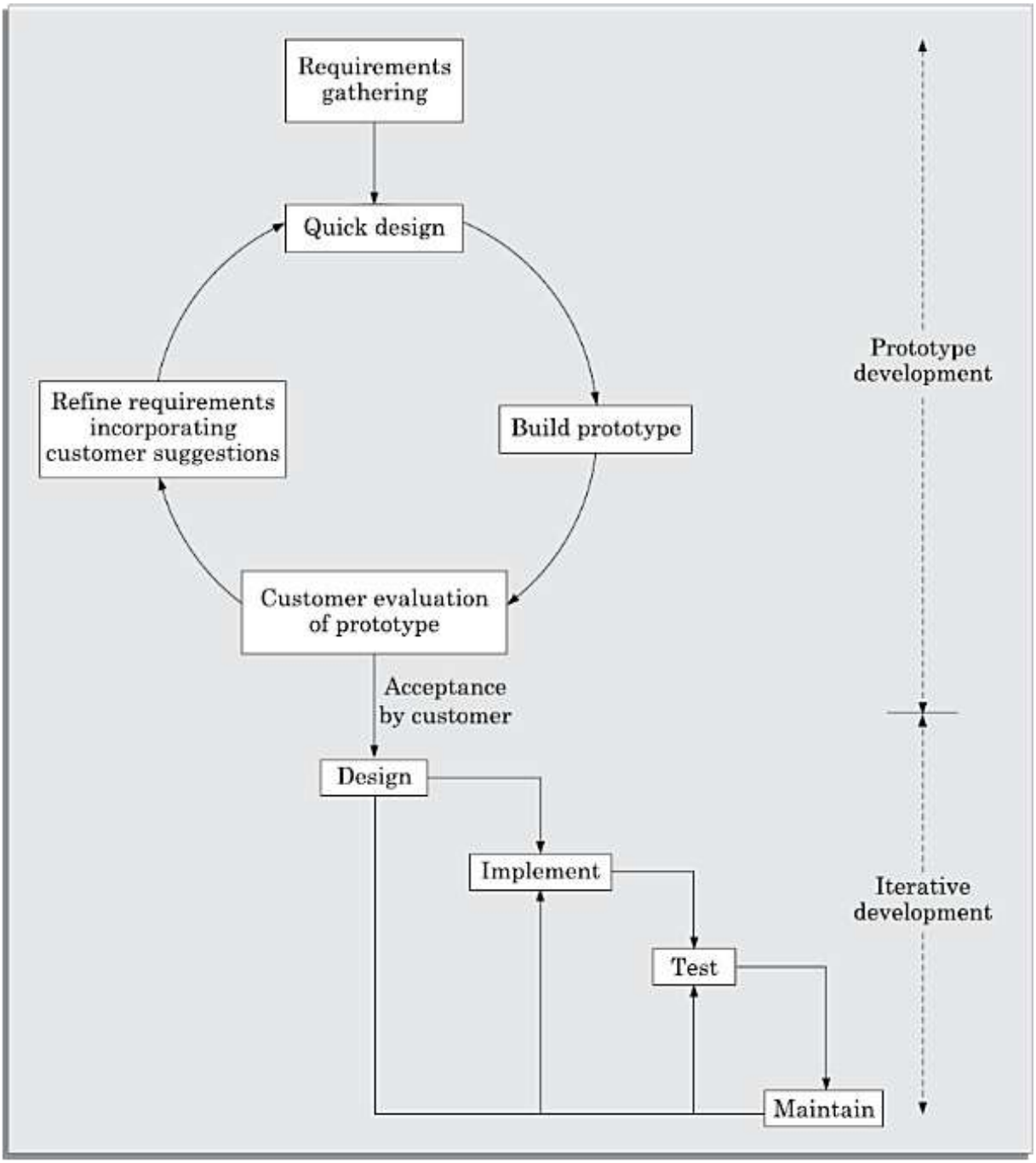


Figure 2.6: Prototyping model of software development.

By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimises later change requests

from the customer and the associated redesign costs.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from technical and requirements risks. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts. Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle. The prototyping model would not be appropriate for projects for which the risks can only be identified after the development is underway.

2.1.5 Incremental Development Model

This life cycle model is sometimes referred to as the *successive versions* model and sometimes as the incremental model. In this life cycle model, first a simple working system implementing only a few basic features is built and delivered to the customer. Over many successive iterations successive versions are implemented and delivered to the customer until the desired system is realised. The incremental development model has been shown in Figure 2.7.

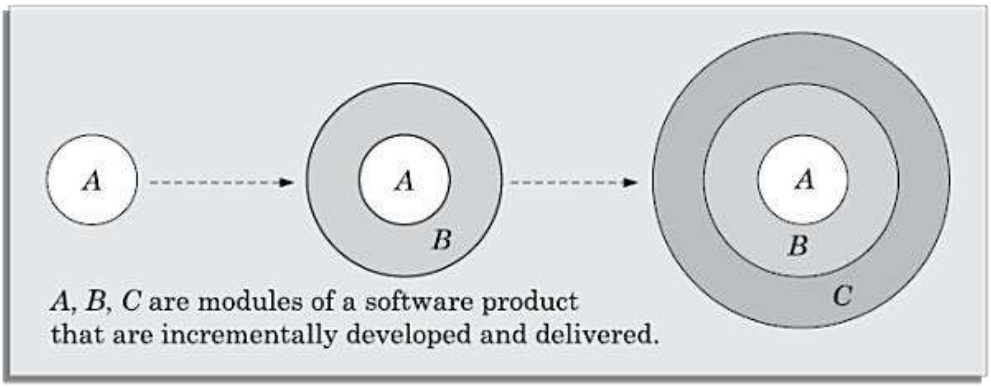


Figure 2.7: Incremental software development.

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted i n Figure 2.7. At any time, plan is made only for the next increment and no long-term plans a re made. Therefore, it becomes easier to accommodate change requests from the customers.

The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in Figure 2.8. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.

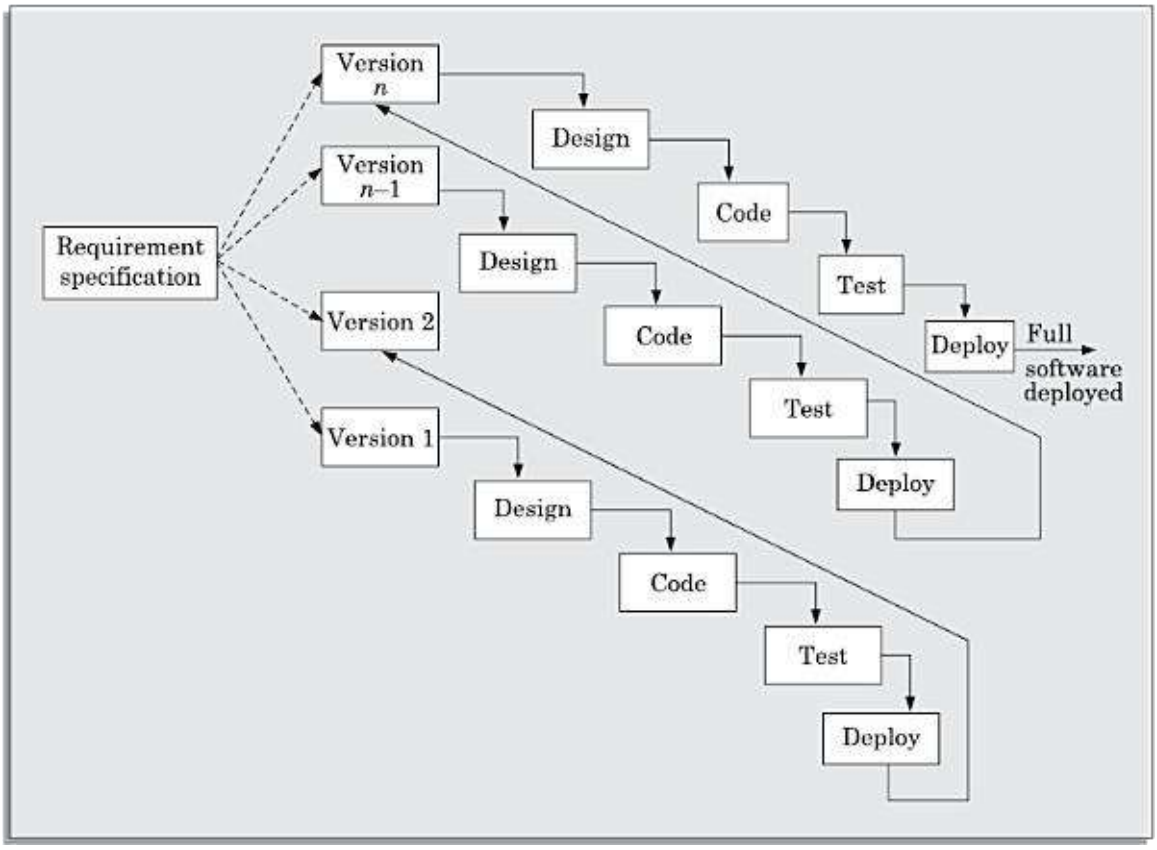


Figure 2.8: Incremental model of software development.

Advantages

The incremental development model offers several advantages. Two important ones are the following:

- **Error reduction:** The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.
- **Incremental resource deployment:** This model obviates the need for the customer to commit large resources at one go for development of the system. It also saves the developing organisation from deploying large resources and manpower for a project in one go.

2.1.6 Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realised. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models. Due to obvious reasons, the evolutionary software development process is sometimes referred to as *design a little, build a little, test a little, deploy a little* model. This means that after the requirements have been specified, the design, build, test, and deployment activities are iterated. A schematic representation of the evolutionary model of development has been shown in Figure 2.9.

Advantages

The evolutionary model of development has several advantages. Two

important advantages of using this model are the following:

- **Effective elicitation of actual customer requirements:** In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.
- **Easy handling change requests:** In this model, handling change requests is easier as no long term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The main disadvantages of the successive versions model are as follows:

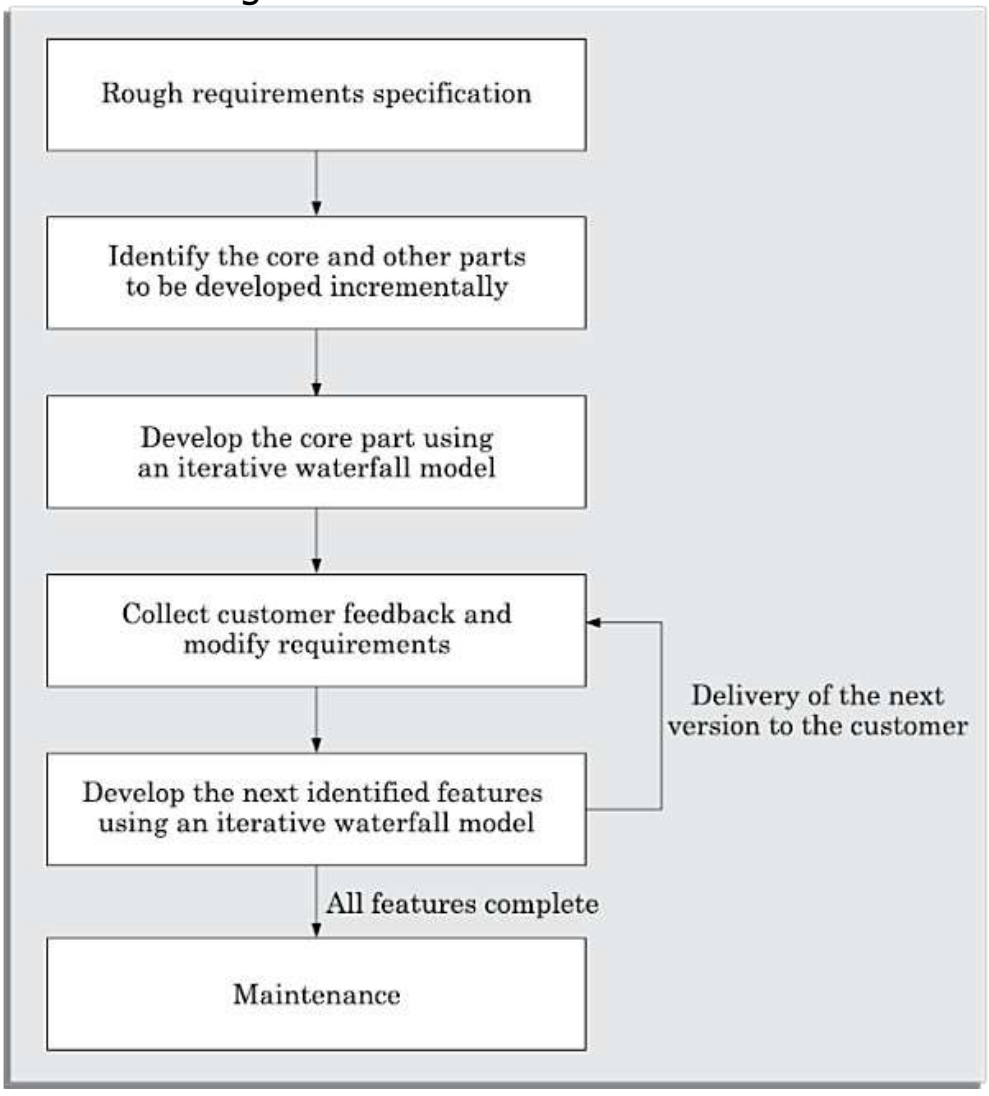


Figure 2.9: Evolutionary model of software development.

- **Feature division into incremental parts can be non-trivial:** For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so interwined and dependent on each other that even an expert would need considerable effort to plan the incremental deliveries.
- ***Ad hoc*** **design:** Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

Applicability of the evolutionary model

The evolutionary model is normally useful for very large products, where it is easier t o find modules for incremental implementation. Often evolutionary model is used when the customer prefers to receive the product in increments so that he can start using the different features as and when they are delivered rather than waiting all the time for the full product to be developed and delivered. Another important category of projects for which the evolutionary model is suitable, is projects using object-oriented development.

The evolutionary model is well-suited to use in object-oriented software development projects.

Evolutionary model is appropriate for object-oriented development project, since it is easy to partition the software into stand alone units in terms of the classes. Also, classes are more or less self contained units that can be developed independently.

2.2 RAPID APPLICATION DEVELOPMENT (RAD)

The *rapid application development* (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the features of both prototyping and evolutionary models. It deploys an evolutionary delivery model to

obtain and incorporate the customer feedbacks on incrementally delivered versions.

In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction

The major goals of the RAD model are as follows:

- To decrease the time taken and the cost incurred to develop software systems.
- To limit the costs of accommodating change requests.
- To reduce the communication gap between the customer and the developers.

Main motivation

In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start. However, often clients do not know what they exactly wanted until they saw a working system. It has now become well accepted among the practitioners that only through the process commenting on an installed application that the exact requirements can be brought out. But in the iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed. Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a *time box*. Each iteration is planned to enhance the implemented functionality of the application by only a small amount. During each time box, a quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback. Please note that the prototype is not meant to be released to the customer for regular use though.

The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team

usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.

RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools to facilitate fast creation of working prototypes. These specialised tools usually support the following features:

- Visual style of development.
- Use of reusable components.

2.2.1 Applicability of RAD Model

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

- **Customised software:** As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software. For example, a company might have developed a software for automating the data processing activities at

one or more educational institutes. When any other institute requests for an automation package to be developed, typically only a few aspects needs to be tailored—since among different educational institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records etc. are similar to a large extent. Projects involving such tailoring can be carried out speedily and cost-effectively using the RAD model.

- **Non-critical software:** The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the Iiterative waterfall model may provide a better solution.
- **Highly constrained pro ject schedule:** RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
- **Large software:** Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

The RAD style of development is not advisable if a development project has one or more of the following characteristics:

- **Generic products (wide distribution):** As we have already pointed out in Chapter 1, software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market. As it has already been discussed, the RAD model of development may not yield systems having optimal performance and reliability.
- **Requirement of optimal performance and/or reliability:** For certain categories of products, optimal performance or reliability is required. Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance

required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

- **Lack of similar products:** If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.
- **Monolithic entity:** For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop a software incrementally.

2.2.2 Comparison of RAD with Other Models

In this section, we compare the relative advantages and disadvantages of RAD with other life cycle models.

RAD *versus* prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away. In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

Though RAD is expected to lead to faster software development compared to the traditional models (such as the prototyping model), though the quality and reliability would be inferior.

RAD *versus* iterative waterfall model

In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the

iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than thatdeveloped using RAD.

RAD *versus* evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

2.3 AGILE DEVELOPMENT MODELS

In the following, a few reasons why the waterfall-based development was becoming difficult to use in project in recent times:

- In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If at all any later requirement changes becomes unavoidable, then the cost of accommodating it becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons.
- Waterfall model is called a *heavy weight* model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.
- Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to

overcome the serious shortcomings of the waterfall model of development identified above. The agile model was primarily designed to help a project to adapt to change requests quickly.[1]Thus, a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project, i.e. removing activities that may not be necessary for a specific project. Also, anything that that wastes time and effort is avoided.

Please note that agile model is being used as an umbrella term to refer to a group of development processes. These processes share certain common characteristics, but do have certain subtle differences among themselves. A few popular agile SDLC models are the following:

- Crystal
- Atern (formerly DSDM)
- Feature-driven development
- Scrum
- Extreme programming (XP)
- Lean development
- Unified process

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasting fo r a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete an iteration is called a *time box*. The implication of the term *time box* is that the end date for an iteration does not change. That is, the delivery date is considered sacrosanct. The development team can, however, decide to reduce the delivered functionality during a time box if necessary.

A central principle of the agile model is the delivery of an increment to the customer after each time box. A few other principles that are central to the agile model are discussed below.

2.3.1 Essential Idea behind Agile Models

For establishing close contact with the customer during development and to gain a clear understanding of the domain-specific issues, each agile project usually includes a customer representative in the team. At the

end of each iteration, stakeholders and the customer representative review the progress made and re-evaluate the requirements. A distinguishing characteristics of the agile models is frequent delivery of software increments to the customer.

Agile model emphasise face-to-face communication over written documents. It is recommended that the development team size be deliberately kept small (5–9 people) to help the team members meaningfully engage in face-to-face communication and have collaborative work environment. It is implicit then that the agile model is suited to the development of small projects. However, if a large project is required to be developed using the agile model, it is likely that the collaborating teams might work at different locations. In this case, the different teams are needed to maintain as much daily contact as possible through video conferencing, telephone, e-mail, etc.

The following important principles behind the agile model were publicised in the agile manifesto in 2001:

- Working software over comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are efficiently incorporated.
- Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process. It is advocated that enhanced communication among the development team members can be realised through face-to-face communication rather than through exchange of formal documents.
- Continuous interaction with the customer is considered much more important rather than effective contract negotiation. A customer representatives is required to be a part of the development team, thus facilitating close, daily co-operation between customers and developers.

Agile development projects usually deploy pair programming.

Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to

programmers working alone.

Advantages and disadvantages of agile methods

The agile methods derive much of their agility by relying on the tacit knowledge of the team members about the development project and informal communications to clarify issues, rather than spending significant amounts of time in preparing formal documents and reviewing them. Though this eliminates some overhead, but lack of adequate documentation may lead to several types of problems, which are as follows:

- Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.
- In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.
- When the project completes and the developers disperse, maintenance can become a problem.

2.3.2 Agile *versus* Other Models

In the following subsections, we compare the characteristics of the agile model with other models of development.

Agile model *versus* iterative waterfall model

The waterfall model is highly structured and systematically steps through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirement specifications, design documents, test plans, code reviews, etc. for which review is complete. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In agile model, delivery of working versions of a software is made in several increments. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

If a project being developed using waterfall model is cancelled mid-way during development, then there i s nothing to show from the abandoned

project beyond several documents. With agile model, even if a project is cancelled midway, it still leaves the customer with some worthwhile code, that might possibly have already been put into live operation.

Agile *versus* exploratory programming

Though a few similarities do exist between the agile and exploratory programming styles, there are vast differences between the two as well. Agile development model's frequent re- evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in exploratory programming.

Agile model *versus* RAD model

The important differences between the agile and the RAD models are the following:

- Agile model does not recommend developing prototypes, but emphasises systematic development of each incremental feature. In contrast, the central theme of RAD is based on designing quick-and-dirty prototypes, which are then refined into production quality code.
- Agile projects logically break down the solution into features that are incrementally developed and delivered. The RAD approach does not recommend this. Instead, developers using the RAD model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.
- Agile teams only demonstrate completed work to the customer. In contrast, RAD teams demonstrate to customers screen mock ups, and prototypes, that may be based on simplifications such as table look-ups rather than actual computations.

2.3.3 Extreme Programming Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by Kent Beck in 1999. The name of this model reflects the fact that it recommends taking these *best practices* that have worked well in the past in program development projects to extreme levels. This model is based on a rather simple

philosophy: "If something is known to be beneficial, why not put it to constant use?" Based on this principle, it puts forward several key practices that need to be practised to the extreme. Please note that most of the key practices that it emphasises were already recognised as good practices for some time.

Good practices that need to be practised to the extreme

In the following subsections, we mention some of the good practices that have been recognised in the extreme programming model and the suggested way to maximise their use:

Code review: It is good since it helps detect and correct problems most efficiently. It suggests *pair programming* as the way to achieve continuous review. In pair programming, coding is carried out by pairs of programmers. The programmers take turn in writing programs and while one writes the other reviews code that is being written.

Testing: Testing code helps to remove bugs and improves its reliability. XP suggests *test-driven development* (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.

Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity makes it easier to develop good quality code, as well as to test and debug it. Therefore, one should try to create the simplest code that makes the basic functionality being written to work. For creating the simplest code, one can ignore the aspects such as efficiency, reliability, maintainability, etc. Once the simplest thing works, other aspects can be introduced through refactoring.

Design: Since having a good quality design is important to develop a good quality solution, everybody should design daily. This can be achieved through *refactoring*, whereby a working code is improved for efficiency and maintainability.

Integration testing: It is important since it helps identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration, by building and performing integration testing several times a day.

Basic idea of extreme programming model

XP is based on frequent releases (called *iteration*), during which the developers implement "user stories". User stories are similar to use cases, but are more informal and are simpler. A user story is the conversational description by the user about a feature of the required system. For example, a user story about a library software can be:

- A library member can issue a book.
- A library member can query about the availability of a book.
- A library member should be able to return a borrowed book.

A user story is a simplistic statement of a customer about a functionality he needs, it does not mention about finer details such as the different scenarios that can occur, the precondition (state at which the system) to be satisfied before the feature can be invoked, etc.

On the basis o f user stories, the project team proposes "metaphors"—a common vision of how the system would work. The development team may decide to construct a *spike* for some feature. A *spike*, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.

X P prescribes several basic activities to be part of the software development process. We discuss these activities in the following subsections:

Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system. Therefore, utmost care and attention need to be placed on coding activity. However, the concept of code as used in XP has a slightly different meaning from what is traditionally understood. For example, coding activity includes drawing diagrams (modelling) that will be transformed to code, scripting a web-based system, and choosing among several alternative solutions.

Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.

Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily be having an in-depth knowledge of the the specific domain of the system under development. On the other hand, customers usually have this domain knowledge. Therefore, for the programmers to properly understand what the

functionality of the system should be, they have to listen to the customer.

Designing: Without proper design, a system implementation becomes too complex and the dependencies within the system become too numerous and it becomes very difficult to comprehend the solution, and thereby making maintenance prohibitively expensive. A good design should result inelimination of complex dependencies within a system. Thus, effective use of a suitable design technique is emphasised.

Feedback: It espouses the wisdom: "A system staying out of users is trouble waiting to happen". It recognises the importance of user feedback in understanding the exact customer requirements. The time that elapses between the development of a version and collection of feedback on it is critical to learning and making changes. It argues that frequent contact with the customer makes the development effective.

Simplicity: A corner-stone of XP is based on the principle: "build something simple that will work today, rather than trying to build something that would take time and yet may never be used". This in essence means that attention should be focused on specific features that are immediately needed and making them work, rather than devoting time and energy on speculations about future requirements.

XP is in favour of making the solution to a problem as simple as possible. In contrast, the traditional system development methods recommend planning for reusability and future extensibility of code and design at the expense of higher code and design complexity.

Applicability of extreme programming model

The following are some of the project characteristics that indicate the suitability of a project for development using extreme programming model:

Projects involving new technology or research pro jects: In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.

Small projects: Extreme programming was proposed in the context of small teams as face to face meeting is easier to achieve.

Project characteristics not suited to development using agile models

The following are some of the project characteristics that indicate

unsuitability of agile development model for use in a development project:

- **Stable requirements:** Conventional development models are more suited to use in projects characterised by stable requirements. For such projects, it is known that few changes, if at all, will occur. Therefore, process models such as iterative waterfall model that involve making long-term plans during project initiation can meaningfully be used.
- **Mission critical or safety critical systems:** In the development of such systems, the traditional SDLC models are usually preferred to ensure reliability.

2.3.4 Scrum Model

In the scrum model, a project is divided into small parts of work that can be incrementally developed and delivered over time boxes that are called *sprints*. The software therefore gets developed over a series of manageable chunks. Each sprint typically takes only a couple of weeks to complete. At the end of each sprint, stakeholders and team members meet to assess the progress made and the stakeholders suggest to the development team any changes needed to features that have already been developed and any overall improvements that they might feel necessary.

In the scrum model, the team members assume three fundamental roles—software owner, scrum master, and team member. The software owner is responsible for communicating the customers vision of the software to the development team. The scrum master acts as a liaison between the software owner and the team, thereby facilitating the development work.

2.4 SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops (see Figure 2.10). The exact number of loops of the spiral is not fixed and can vary from project to project. The number of loops shown in Figure 2.10 is just an example. Each loop of the spiral is called a *phase* of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks. A prominent feature of the spiral model is handling

unforeseen risks that can show up much after the project has started. In this context, please recollect that the prototyping model can be used effectively only when the risks in a project can be identified upfront before the development work starts. As we shall discuss, this model achieves this by incorporating much more flexibility compared to SDLC other models.

While the prototyping model does provide explicit support for risk handling, the risks are assumed to have been identified completely before the project start. This is required since the prototype is constructed only at the start of the project. In contrast, in the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation. Over each loop, one or more features of the product are elaborated and analysed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

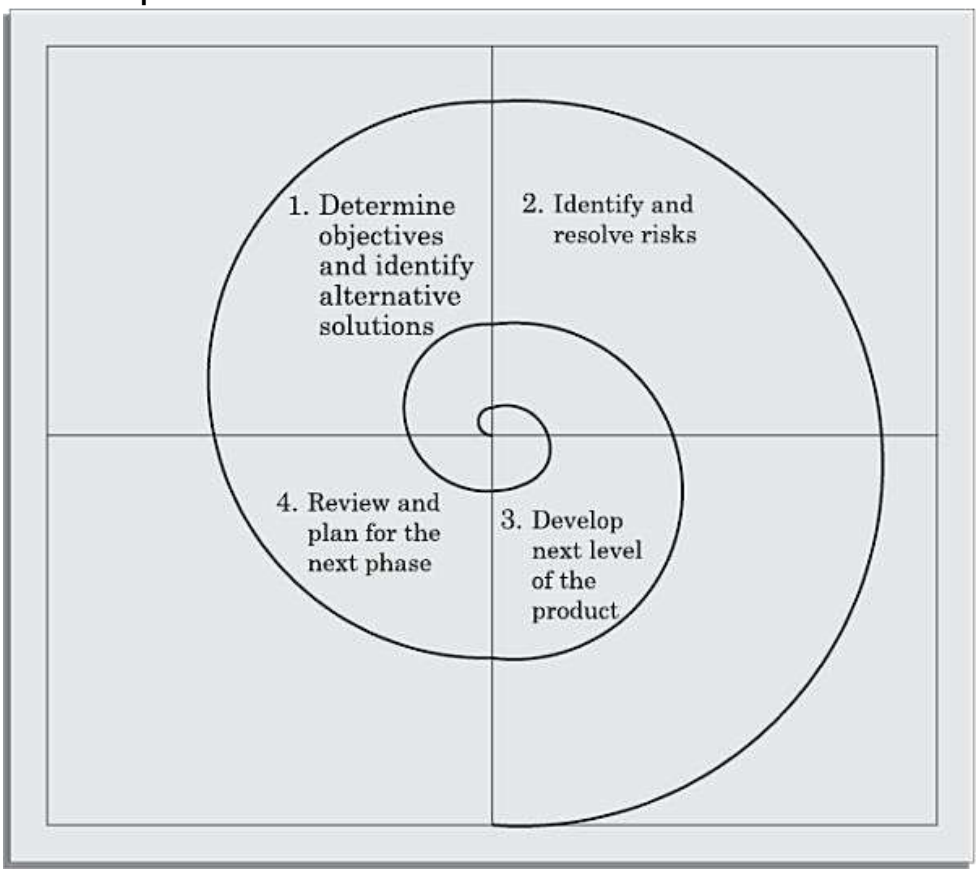


Figure 2.10: Spiral model of software development.

Risk handling in spiral model

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database

might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a faster communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

2.4.1 Phases of the Spiral Model

Each phase in this model is split into four sectors (or quadrants) as shown in Figure 2.10. In the first quadrant, a few features of the software are identified to be taken u p for immediate development based on how crucial it is to the overall software development. With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.

The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so

far in the current phase.

In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model tospecific projects.

To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified. To keep our discussion simple, we shall not delve into parallel cycles in the spiral model.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to a only a few types of projects. To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk-driven and is more complicated phase structure than the other models we discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project. Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed.

In spite of the disadvantages of the spiral model that we pointed out, for certain categories of projects, the advantages of the spiral model can outweigh its disadvantages.

In this regard, it is much more powerful than the prototyping model. Prototyping model can meaningfully be used when all the risks associated with a project are known beforehand. All these risks are resolved by building a prototype before the actual software development starts.

Spiral model as a meta model

As compared to the previously discussed models, the spiral model can be viewed as a *meta model*, since it subsumes all the discussed models. For example, a single loop spiral actually represents the waterfall model. The spiral model uses the approach of the prototyping model by first building a prototype in each phase before the actual development starts. This prototypes are used as a risk reduction mechanism. The spiral model incorporates the systematic step- wise approach of the waterfall model. Also, the spiral model can be considered as supporting the evolutionary model—the

iterations along the spiral can be considered as evolutionary levels through which the complete system is built. This enables the developer to understand and resolve the risks at each evolutionary level (i.e. iteration along the spiral).

2.5 A COMPARISON OF DIFFERENT LIFE CYCLE MODELS

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to correct the errors that are committed during any of the phases but detected at a later phase. This problem is overcome by the iterative waterfall model through the provision of feedback paths.

The iterative waterfall model is probably the most widely used software development model so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems, and is not suitable for development of very large projects and projects that suffer from large number of risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood, however all the risks can be identified before the project starts. This model is especially popular for development of the user interface part of projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also used widely for object-oriented development projects. Of course, this model can only be used if incremental delivery of the system is acceptable to the customer.

The spiral model is considered a *meta model* and encompasses all other life cycle models. Flexibility and risk handling are inherently built into this model. The spiral model is suitable for development of technically challenging and large software that are prone to several kinds of risks that are difficult to anticipate at the start of the project. However, this model is mu ch more complex than the other models—this is probably a factor deterring its use in ordinary projects.

Let us now compare the prototyping model with the spiral model. The prototyping model can be used if the risks are few and can be determined at the start of the project. The spiral model, on the other hand, is useful when

the risks are difficult to anticipate at the beginning of the project, but are likely to crop up as the development proceeds.

Let us compare the different life cycle models from the viewpoint of the customer. Initially, customer confidence is usually high on the development team irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working software is yet visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working software much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the software via incremental phases provides time to the customer to adjust to the new software. Also, from the customer's financial view point, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

2.5.1 Selecting an Appropriate Life Cycle Model for a Pro ject

We have discussed the advantages and disadvantages of the various life cycle models. However, how to select a suitable life cycle model for a specific project? The answer to this question would depend on several factors. A suitable life cycle model can possibly be selected based on an analysis of issues such as the following:

Characteristics of the software to be developed: The choice of the life cycle model to a large extent depends on the nature of the software that is being developed. For small services projects, the agile model is favoured. On the other hand, for product and embedded software development, the iterative waterfall model can be preferred. An evolutionary model is a suitable model for object-oriented development projects.

Characteristics of the development team: The skill-level of the team members is a significant factor in deciding about the life cycle model to use. If the development team is experienced in developing similar software, then even an embedded software can be developed using an iterative waterfall model. If the development team is entirely novice, then even a simple data processing application may require a prototyping model to be adopted.

Characteristics of the customer: If the customer is not quite familiar with computers, then the requirements are likely to change frequently as it would

be difficult to form complete, consistent, and unambiguous requirements. Thus, a prototyping model may be necessary to reduce later change requests from the customers.

Chapter

3

SOFTWARE PROJECT MANAGEMENT

3.1 SOFTWARE PROJECT MANAGEMENT COMPLEXITIES

Management of software projects is much more complex than management of many other types of projects. The main factors contributing to the complexity of managing a software project, as identified by [Brooks75], are the following:

Invisibility: Software remains invisible, until its development is complete and it is operational. Anything that is invisible, is difficult to manage and control. Consider a house building project. For this project, the project manger can very easily assess the progress of the project through a visual examination of the building under construction. Therefore, the manager can closely monitor the progress of the project, and take remedial actions whenever he finds that the progress is not as per plan. In contrast, it becomes very difficult for the manager of a software project to assess the progress of the project due to the invisibility of software. The best that he can do perhaps is to monitor the milestones that have been completed by the development team and the documents that have been produced—which are rough indicators of the progress achieved.

Changeability: Because the software part of any system is easier to change as compared to the hardware part, the software part is the one that gets most frequently changed. This is especially true in the later stages of a project. As far as hardware development is concerned, any late changes to the specification of the hardware system under development usually amounts to redoing the entire project. This makes late changes to a hardware project prohibitively expensive to carry out. This possibly is a reason why requirement changes are frequent in software projects. These changes usually arise from changes to the business practices, changes to the hardware or underlying software (e.g. operating system, other applications), or just because the client changes his mind.

Complexity: Even a moderate sized software has millions of parts (functions) that interact with each other in many ways—data coupling, serial and concurrent runs, state transitions, control dependency, file sharing, etc. Due to the inherent complexity of the functioning of a software product in terms of the basic parts making up the software, many types of risks are associated with its development. This makes managing these projects much more difficult as compared to many other kinds of projects.

Uniqueness: Every software project is usually associated with many unique features or situations. This makes every project much different from the others. This is unlike projects in other domains, such as car manufacturing or steel manufacturing where the projects are more predictable. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the others he had encountered in the past. As a result, a software project manager has to confront many unanticipated issues in almost every project that he manages.

Exactness of the solution: Mechanical components such as nuts and bolts typically work satisfactorily as long as they are within a tolerance of 1 per cent or so of their specified sizes. However, the parameters of a function call in a program are required to be in complete conformity with the function definition. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult. This requirement of exact conformity of the parameters of a function introduces additional risks and contributes to the complexity of managing software projects.

Team-oriented and intellect-intensive work: Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work. In contrast, projects in many domains are labour-intensive and each member works in a high degree of autonomy. Examples of such projects are planting rice, laying roads, assembly-line manufacturing, constructing a multi-storeyed building, etc. In a software development project, the life cycle activities not only highly intellect-intensive, but each member has to typically interact, review, and interface with several other members, constituting another dimension of complexity of software projects.

3.2 RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

In this section, we examine the principal job responsibilities of a project manager and the skills necessary to accomplish those.

3.2.1 Job Responsibilities for Managing Software Projects

A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description. In fact, it is very difficult to objectively describe the precise job responsibilities of a project manager. The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations. Most managers take the responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients. These activities are certainly numerous and varied. We can still broadly classify these activities into two major types—project planning and project monitoring and control.

In the following subsections, we give an overview of these two classes of responsibilities. Later on, we shall discuss them in more detail.

Project planning: Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.

The initial project plans are revised from time to time as the project progresses and more project data become available.

Project monitoring and control: Project monitoring and control activities are undertaken once the development activities start. While carrying out project monitoring and control activities, a project manager may sometimes find it necessary to change the plan to cope up with specific situations at hand.

3.2.2 Skills Necessary for Managing Software Projects

A theoretical knowledge of various project management techniques is certainly important to become a successful project manager. However, a purely theoretical knowledge of various project management techniques would hardly make one a successful project manager. Effective software project management calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project

management techniques such as cost estimation, risk management, and configuration management, etc., project managers need good communication skills and the ability to get work done. Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. Never the less, the importance of a sound knowledge of the prevalent project management techniques cannot be overemphasized. The objective of the rest of this chapter is to introduce the reader to the same.

Three skills that are most critical to successful project management are the following:
- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

With this brief discussion on the overall responsibilities of a software project manager and the skills necessary to accomplish these, in the next section we examine some important issues in project planning.

3.3 METRICS FOR PROJECT SIZE ESTIMATION

As already mentioned, accurate estimation of project size is central to satisfactory estimation of all other project parameters such as effort, completion time, and total project cost. Before discussing the available metrics to estimate the size of a project, let us examine what does the term "project size" exactly mean. The size of a project is obviously not the number of bytes that the source code occupies, neither is it the size of the executable code.

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages. These are discussed in the following subsection. Based on their relative advantages, one metric may be more appropriate than the other in a particular situation.

3.3.1 Lines of Code (LOC)

LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are

ignored.

Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task. One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and so on, until the LOC of the leaf-level modules are small enough to be predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar modules is very helpful. By adding the estimates for all leaf level modules together, project managers arrive at the total size estimation. In spite of its conceptual simplicity, LOC metric has several shortcomings when used to measure problem size. We discuss the important shortcomings of the LOC metric in the following subsections:

LOC is a measure of coding activity alone. A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, design, code, test, etc.) and not just the coding effort. LOC, however, focuses on the coding activity alone—it merely computes the number of source lines in the final program. We have already discussed in Chapter 2 that coding is only a small part of the overall software development effort.

The presumption that the total effort needed to develop a project is proportional to the coding effort is easily countered by noting the fact that even when the design or testing issues are very complex, the code size might be small and vice versa. Thus, the design and testing efforts can be grossly disproportional to the coding effort. Code size, therefore, is obviously an improper indicator of the problem size.

LOC count depends on the choice of specific instructions: LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers. By coding style, we mean the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used. Different programmers may lay out their code in very different ways. For example, one programmer might write several source instructions on a single line, whereas another might split a single instruction across several lines. Unless this issue is handled satisfactorily, there is a possibility of arriving at very different size measures for essentially identical programs. This problem can, to a large extent, be overcome by counting the

language tokens in a program rather than the lines of code. However, a more intricate problem arises due to the specific choices of instructions made in writing the program. For example, one programmer may use a switch statement in writing a C program and another may use a sequence of if ... then ... else ... statements. Therefore, the following can easily be concluded.

LOC measure correlates poorly with the quality and efficiency of the code: Larger code size does not necessarily imply better quality of code or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms. In fact, it is true that a piece of poor a n d sloppily written piece of code can have larger number of source instructions than a piece t h a t is efficient and has been thoughtfully written. Calculating productivity as LOC generated per man-month may encourage programmers to write lots of poor quality code rather than fewer lines of high quality code achieve the same functionality.

LOC metric penalises use of higher-level programming languages and code reuse: A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would sh ow up as smaller program size, and in turn, would indicate lower effort! Thus, if managers use the LOC count to measure the effort put in by different developers (that is, their productivity), they would be discouraging code reuse by developers. Modern programming methods such as object-oriented programming and reuse of components makes the relationships between LOC and other project attributes even less precise.

LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities: Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic. To realise why this is so, imagine the effort that would be required to develop a program having multiple nested loops and decision constructs and compare that with another program having only sequential control flow.

It is very difficult to accurately estimate LOC of the final program from problem specification: As already discussed, at the project initiation time, it is a very difficult task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity

starts, the LOC metric is of little use to the project managers during project planning.

3.3.2 Function Point (FP) Metric

Function point metric was proposed by Albrecht in 1983. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has steadily gained popularity. Function point metric has several advantages over LOC metric. One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself. Using the LOC metric, on the other hand, the size can accurately be determined only after the product has fully been developed.

The conceptual idea behind the function point metric is the following. The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.

Though each feature takes some effort to develop, different features may take very different amounts of efforts to develop. For example, in a banking software, a function to display a help message may be much easier to develop compared to say the function that carries out the actual banking transactions. This has been considered by the function point metric by counting the number of input and output data items and the number of files accessed by the function.

The implicit assumption made is that the more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function. Now let us analyse why this assumption must be intuitively correct. Each feature when invoked typically reads some input data and then transforms those to the required output data. For example, the query book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. It can therefore be argued that the computation of the number of input and output data items

would give a more accurate indication of the code size compared to simply counting the number of high-level functions supported by the system.

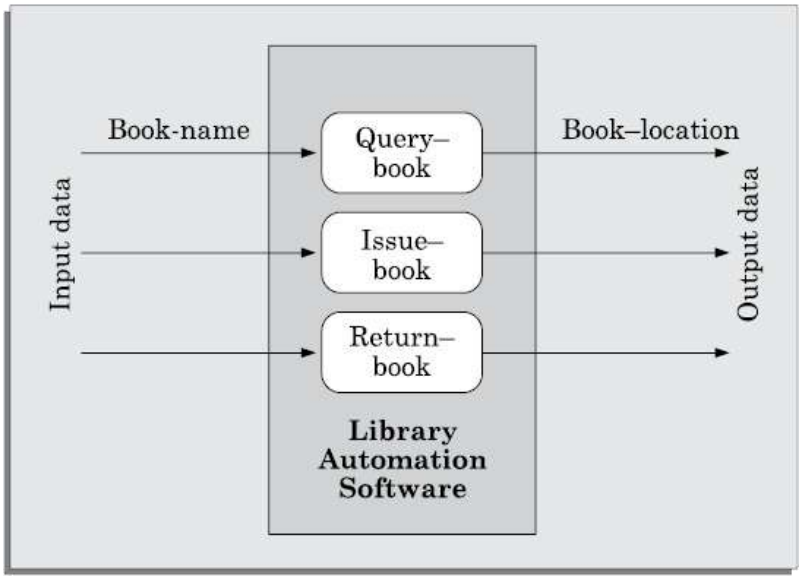


Figure 3.2: System function as a mapping of input data to output data.

Albrecht postulated that in addition to the number of basic functions that a software performs, size also depends on the number of files and the number of interfaces that are associated with the software. Here, interfaces refer to the different mechanisms for data transfer with external systems including the interfaces with the user, interfaces with external computers, etc.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

- **Step 1:** Compute the unadjusted function point (UFP) using a heuristic expression.
- **Step 2:** Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
- **Step 3:** Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

We discuss these three steps in more detail in the following.

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of five characteristics of a product as shown in the following expression. The weights associated with the five characteristics were determined empirically by Albrecht through data gathered from many projects.

```
UFP = (Number of inputs)*4 + (Number of outputs)*5 +
 (Number of inquiries)*4 + (Number of files)*10 +
 (Number of interfaces)*10                    (3.1)
```

The meanings of the different parameters of Eq. 3.1 are as follows:

1. **Number of inputs:** Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries. Inquiries are user commands such as print-account-balance that require no data values to be input by the user. Inquiries are counted separately (see the third point below). It needs to be further noted that individual data items input by the user are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input. For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they describe a single employee.
2. **Number of outputs:** The outputs considered include reports printed, screen outputs, error messages produced, etc. While computing the number of outputs, the individual data items within a report are not considered; but a set of related data items is counted as just a single output.
3. **Number of inquiries:** An inquiry is a user command (without any data input) and only requires some actions to be performed by the system. Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users. Examples of such inquiries are print account balance, print all student grades, display rank holders' names, etc.
4. **Number of files:** The files referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.
5. **Number of interfaces:** Here the interfaces denote the different mechanisms that are used to exchange information with other

systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

Step 2: Refine parameters

UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined. This is possible, since each parameter (input, output, etc.) has been implicitly assumed to be of average complexity. However, this is rarely true. For example, some input values may be extremely complex, some very simple, etc. In order to take this issue into account, UFP is refined by taking into account the complexities of the parameters of UFP computation (Eq. 3.1). The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for the different parameters are determined based on the numerical values shown in Table 3.1. Based on these weights of the parameters, the parameter values in the UFP are refined. For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

Table 3.1: Refinement of Function Point Entities

Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2. Examples of such project parameters that can influence the project sizes include high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort. The list of these parameters have been shown in Table 3.2. Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of the corresponding project parameters on the development effort. TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 84, TCF can vary from

0.65 to 1.49. Finally, FP is given as the product of UFP and TCF. That is, FP=UFP*TCF.

Table 3.2: Function Point Relative Complexity Adjustment Factors
Requirement for reliable backup and recovery
Requirement for data communication
Extent of distributed processing
Performance requirements
Expected operational environment
Extent of online data entries
Extent of multi-screen or multi-operation online data input
Extent of online updating of master files
Extent of complex inputs, outputs, online queries and files
Extent of complex data processing
Extent that currently developed code can be designed for reuse
Extent of conversion and installation included in the design
Extent of multiple installations in an organisation and variety of customer organisations
Extent of change and focus on ease of use

Example 3.1 Determine the function point measure of the size of the following supermarket software. A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Ea ch customer who registers for this scheme is assigned a unique customer number (CN) by the computer. Based on the generated CN, a clerk manually prepares a customer identity card after getting the market manager's signature on it. A customer can present his customer identity card to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs. 10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated. Assume that various project characteristics determining the complexity of software development to be average.

Answer:

Step 1: From an examination of the problem description, we find that there are two inputs, three outputs, two files, and no interfaces. Two files would be required, one for storing the customer details and another for storing the daily purchase records. Now, using equation 3.1, we get:

$$UFP = 2 \times 4 + 3 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 47$$

Step 2: All the parameters are of moderate complexity, except the

output parameter of customer registration, in which the only output is the CN value. Consequently, the complexity of the output parameter of the customer registration function can be categorized as simple. By consulting Table 3.1, we find that the value for simple output is given to be 4. The UFP can be refined as follows:

$$UFP = 3 \times 4 + 2 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 46$$

Therefore, the UFP will be 46.

Step 3: Since the complexity adjustment factors have average values, therefore the total degrees of influence would be: $DI = 14 \times 4 = 56$

$$TCF = 0.65 + 0.01 + 56 = 1.21$$

Therefore, the adjusted FP=46*1.21=55.66

Feature point metric shortcomings: A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a function. That is, the function point metric implicitly assumes that the effort required to design and develop any two different functionalities of the system is the same. But, we know that this is highly unlikely to be true. The effort required to develop any two functionalities may vary widely. For example, in a library automation software, the create-member feature would be much simpler compared to the loan-from-remote-library feature. FP only considers the number of functions that the system supports, without distinguishing the difficulty levels of developing the various functionalities. To overcome this problem, an extension to the function point metric called feature point metric has been proposed.

Feature point metric incorporates algorithm complexity as an extra parameter. This parameter ensures that the computed size using the feature point metric reflects the fact that higher the complexity of a function, the greater the effort required to develop it—therefore, it should have larger size compared to a simpler function.

Critical comments on the function point and feature point metrics

Proponents of function point and feature point metrics claim that these t w o metrics are language-independent and can be easily computed from the SRS document during project planning stage itself. On the other hand, opponents claim that these metrics are subjective and require a sleight of hand. An example of the subjective nature of the

function point metric can be that the way one groups input and output data items into logically related groups can be very subjective. For example, consider that certain functionality requires the employee name and employee address to be input. It is possible that one can consider both these items as a single unit of data, since after all, these describe a single employee. It is also possible for someone else to consider an employee's address as a single unit of input data and name as another. Such ambiguities leav e sufficient scope for debate and keep open the possibility for different project managers to arrive at different function point measures for essentially the same problem.

3.4 PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost. Accurate estimation of these parameters is important, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling. A large number of estimation techniques have been proposed by researchers. These can broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

In the following subsections, we provide an overview of the different categories of estimation techniques.

3.4.1 Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalised to a large extent. We shall discuss two such formalisations of the basic empirical estimation techniques known as expert judgement and the Delphi techniques in Sections 3.6.1 and 3.6.2 respectively.

3.4.2 Heuristic Techniques

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

$$\text{Estimated Parameter} = c_1 \square e^{d_1}$$

In the above expression, e represents a characteristic of the software that has already been estimated (independent variable). Estimated P arameter is the dependent parameter (to be estimated). The dependent parameter to be estimated could be effort, project duration, staff size, etc., c_1 a nd d_1 are constants. The values of the constants c_1 a nd d_1 a re usually determined using data collected from past projects (historical data). The COCOMO model discussed in Section 3.7.1, is an example of a single variable cost estimation model.

A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

$$\text{Estimated Resource} = c_1 \square p_1^{d_1} + c_2 \square p_2^{d_2} + \ldots$$

where, p_1, p_2, ... are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, ... are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants c_1 ,

d_1 , c_2 , d_2 , ... Values of these constants are usually determined from an analysis of historical data. The intermediate COCOMO model discussed in Section 3.7.2 can b e considered to be an example of a multivariable estimation model.

3.4.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project. Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. As an example of an analytical technique, we shall discuss the Halstead's software science in Section 3.8. We shall see that starting with a few simple assumptions, Halstead's software science derives some interesting results. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned.

3.5 EMPIRICAL ESTIMATION TECHNIQUES

We have already pointed out that empirical estimation techniques have, over the years, been formalised to a certain extent. Yet, these are still essentially euphemisms for pure guess work. These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are—Expert judgement and Delphi estimation techniques. We discuss these two techniques in the following subsection.

3.5.1 Expert Judgement

Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly.

Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant

experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts, but may not be very knowledgeable about the computer communication part. Due to these factors, the size estimation arrived at by the judgement of a single expert may be far from being accurate.

A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimised when the estimation is done by a group of experts. However, the estimate made by a group of experts may still exhibit bias. For example, on certain issues the entire group of experts may be biased due to reasons such as those arising out of political or social considerations. Another important shortcoming of the expert judgement technique is that the decision made by a group may be dominated by overly assertive members.

3.5.2 Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator. In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the co-ordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process. The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate. The Delphi estimation, though consumes more time and effort,

overcomes an important shortcoming of the expert judgement technique in that the results can not unjustly be influenced by overly assertive and senior members.

3.6 COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COnstructive COst estimation MOdel (COCOMO) was proposed by Boehm [1981]. COCOMO prescribes a three stage process for project estimation. In the first stage, an initial estimate is arrived at. Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate. COCOMO uses both single and multivariable estimation models at different stages of estimation.

The three stages of COCOMO estimation technique are—basic COCOMO, intermediate COCOMO, and complete COCOMO. We discuss these three stages of estimation in the following subsection.

3.6.1 Basic COCOMO Model

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—organic, semidetached, and embedded. Based on the category of a software development project, he gave different sets of formulas to estimate the effort and duration from the size estimate.

Three basic classes of software development projects

In order to classify a project into the identified categories, Boehm requires us to consider not only the characteristics of the product but also those of the development team and development environment. Roughly speaking, the three product development classes correspond to development of application, utility and system software. Normally, data processing programs[1] are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and programming complexities also arise out of the requirement for meeting timing constraints and concurrent processing of tasks.

Brooks [1975] states that utility programs are roughly three times as difficult to write as application programs and system programs are roughly three times as difficult as utility programs. Thus according to Brooks, the

relative levels of product development complexity for the three categories (application, utility and system programs) of products are 1:3:9.

Boehm's [1981] definitions of organic, semidetached, and embedded software are elaborated as follows:

Organic: We can classify a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be classify to be of semidetached type, if the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Observe that in deciding the category of the development project, in addition to considering the characteristics of the product being developed, we need to consider the characteristics of the team members. Thus, a simple data processing program may be classified as semidetached, if the team members are inexperienced in the development of similar products.

For the three product categories, Boehm provides different sets of expressions to predict the effort (in units of person-months) and development time from the size estimation given in kilo lines of source code (KLSC). But, how much effort is one person-month?

> One person month is the effort an individual can typically put in a month. The person-month estimate implicitly takes into account the productivity losses that normally occur due to time lost in holidays, weekly offs, coffee breaks, etc.

What is a person-month?

Person-month (PM) is a popular unit for effort measurement.

> Person-month (PM) is considered to be an appropriate unit for measuring effort, because developers are typically assigned to a project for a certain number of months.

It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither does it imply that 1 person should be employed for 100 months to complete the project. The effort estimation simply denotes the area under the person-month curve (see Figure 3.3) for the project. The plot in Figure 3.3 shows that different number of personnel may work at different points in the project development. The number of personnel working on the project usually increases or decreases by an integral number, resulting in the sharp edges in the plot. We shall elaborate in Section 3.9 how the exact number of persons to work at any time on the product development can be determined from the effort and duration estimates.

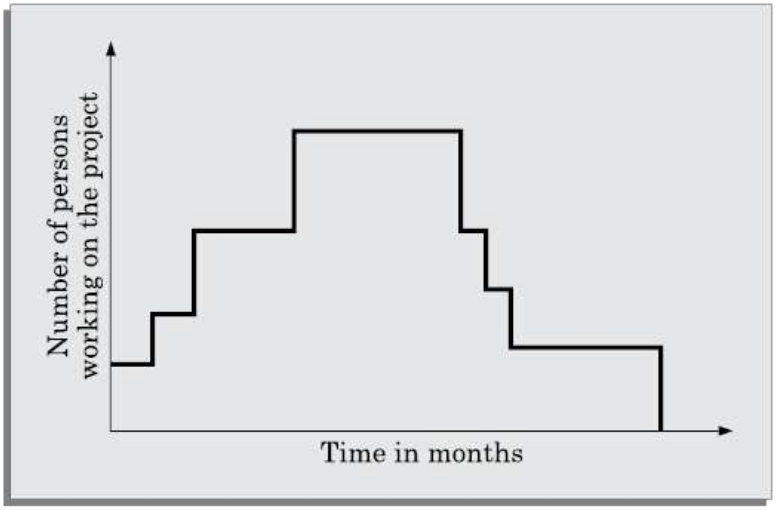


Figure 3.3: Person-month curve.

General form of the COCOMO expressions

The **basic COCOMO model** is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ months}$$

where,

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- a_1, a_2, b_1, b_2 are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in

months.

- Effort is the total effort required to develop the software product, expressed in person- months (PMs).

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products as given by Boehm [1981] are summarised below. He derived these values by examining historical data collected from a large number of actual projects.

Estimation of development effort: For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = 2.4(KLOC)1.05 PM

Semi-detached : Effort = 3.0(KLOC)1.12 PM

Embedded : Effort = 3.6(KLOC)1.20 PM

Estimation of development time: For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort)0.38 Months

Semi-detached : Tdev = 2.5(Effort)0.35 Months

Embedded : Tdev = 2.5(Effort)0.32 Months

We can gain some insight into the basic COCOMO model, if we plot the estimated effort and duration values for different software sizes. Figure 3.4 shows the plots of estimated effort versus product size for different categories of software products.

Observations from the effort-size plot From Figure 3.4, we can observe that the effort is some what superlinear (that is, slope of the curve>1) in the size of the software product.

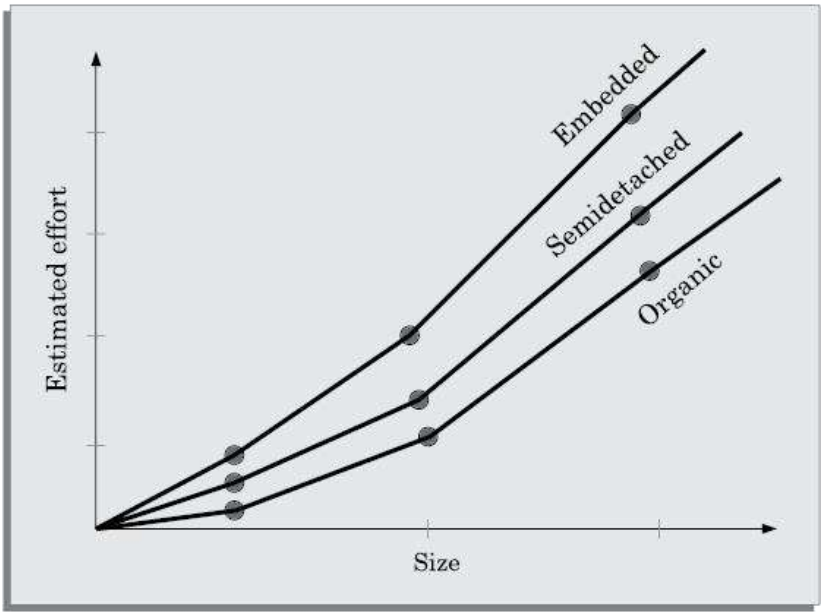


Figure 3.4: Effort versus product size.

This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. However, observe that the increase in effort with size is not as bad as that was portrayed in Chapter 1. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles.

Observations from the development time—size plot

The development time versus the product size in KLOC is plotted in Figure 3.5. From Figure 3.5, we can observe the following:

- The development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately. For example, to develop a product twice as large as a product of size 100KLOC, the increase in duration may only be 20 per cent. It may appear surprising that the duration curve does not increase superlinearly—one would normally expect the curves to behave similar to those in the effort-size plots. This apparent anomaly can be explained by the fact that COCOMO assumes that a project development is carried out not by a single person but by a team of developers.
- From Figure 3.5 we can observe that for a project of any given size, the

development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semi-detached, or embedded type. (Please verify this using the basic COCOMO formulas discussed in this section). However, according to the COCOMO formulas, embedded programs require much higher effort than either application or utility programs. We can interpret it to mean that there is more scope for parallel activities for system programs than those in utility or application programs.

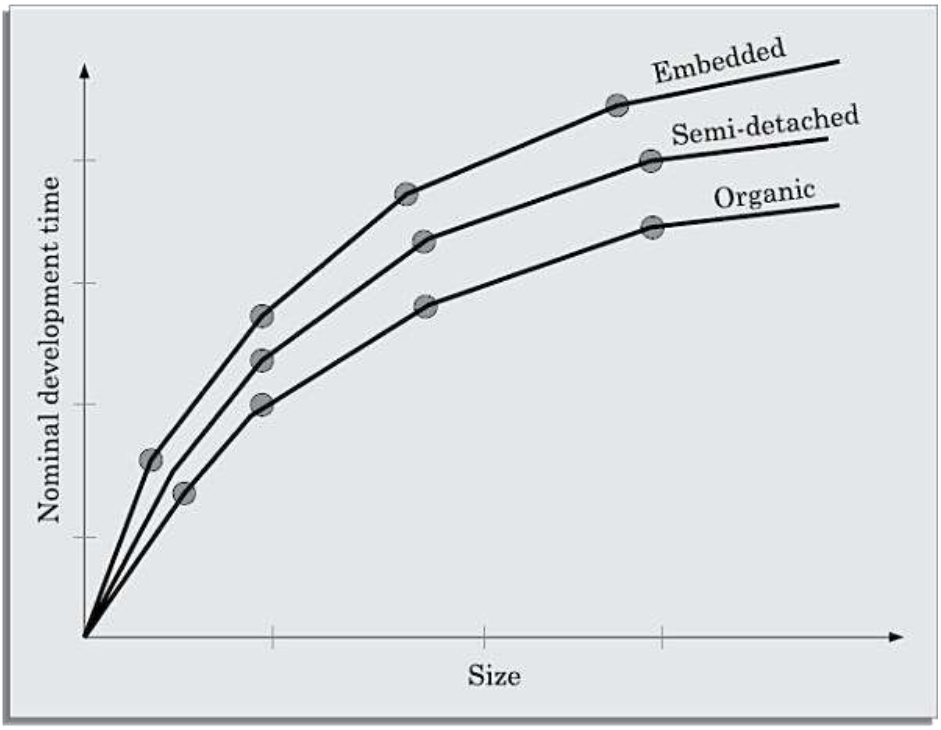


Figure 3.5: Development time versus size.

Cost estimation

From the effort estimation, project cost can be obtained by multiplying the estimated effort (in man-month) by the manpower cost per month. Implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. However, in addition to manpower cost, a project would incur several other types of costs which we shall refer to as the overhead costs. The overhead costs would include the costs due to hardware and software required for the project and the company overheads for administration, office space,electricity, etc. Depending on the expected values of the overhead costs, the project manager has to suitably scale up the cost

arrived by using the COCOMO formula.

Implications of effort and duration estimate

An important implicit implication of the COCOMO estimates are that if you try to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if you complete the project over a longer period of time than that estimated, then there is almost no decrease in the estimated cost value. The reasons for this are discussed in Section 3.9. Thus, we can consider that the COCOMO effort and duration values to indicate the following.

Staff-size estimation

Given the estimations for the project development effort and the nominal development time, can the required staffing level be determined by a simple division of the effort estimation by the duration estimation? The answer is "No". It will be a perfect recipe for project delays and cost overshoot. We examine the staffing problem in more detail in Section 3.9. From the discussions in Section 3.9, it would become clear that the simple division approach to obtain the staff size would be highly improper.

Example 3.2 Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is Rs. 15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

From the basic COCOMO estimation formula for organic software: Effort = $2.4 \times (32)^{1.05} = 91$ PM

Nominal development time = $2.5 \times (91)^{0.38} = 14$ months

Staff cost required to develop the product = $91 \times$ Rs. 15, 000 = Rs. 1,465,000

3.6.2 Intermediate COCOMO

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort as well as the time required to develop the product. For example the effort to develop a product would vary depending upon the sophistication of the

development environment.

Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognises this fact and refines the initial estimates.

The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, the initial estimates are scaled upward. Boehm requires the project manager to rate 15 different parameters for a particular project on a scale of one to three. For each such grading of a project parameter, he has suggested appropriate cost drivers (or multipliers) to refine the initial estimates.

In general, the cost drivers identified by Boehm can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

Development environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

We have discussed only the basic ideas behind the intermediate COCOMO model. A detailed discussion on the intermediate COCOMO model are beyond the scope of this book and the interested reader may refer [Boehm81].

3.6.3 Complete COCOMO

A major shortcoming of both the basic and the intermediate COCOMO models is that they consider a software product as a single

homogeneous entity. However, most large systems are made up of several smaller sub-systems. These sub-systems often have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some even embedded. Not only may the inherent development complexity of the subsystems be different, but for some subsystem the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on.

The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual sub-systems.

In other words, the cost to develop each sub-system is estimated separately, and the complete system cost is determined as the subsystem costs. This approach reduces the margin of error in the final estimate.

Let us consider the following development project as an example application of the complete COCOMO model. A distributed management information system (MIS) product for an organisation having offices at several places across the country can have the following sub-component:

- Database part
- Graphical user interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

To further improve the accuracy of the results, the different parameter values of the model can be fine-tuned and validated against an organisation's historical project database to obtain more accurate estimations. Estimation models such as COCOMO are not totally accurate and lack a full scientific justification. Still, software cost estimation models such as COCOMO are required for an engineering approach to software project management. Companies consider computed cost estimates to be satisfactory, if these are within about 80 per cent of the final cost. Although these estimates are gross approximations—without such models, one has only subjective judgements to rely on.

3.6.4 COCOMO 2

Since the time that COCOMO estimation model was proposed in the early

1980s, the software development paradigms as well as the characteristics of development projects have undergone a sea change. The present day software projects are much larger in size and reuse of existing software to develop new products has become pervasive. For example, component-based development and service-oriented architectures (SoA) have become very popular (discussed in Chapter 15). New life cycle models and development paradigms are being deployed for web-based and component-based software. During the 1980s rarely any program was interactive, and graphical user interfaces were almost non-existent. On the other hand, the present day software products are highly interactive and support elaborate graphical user interface. Effort spent on developing the GUI part is often as much as the effort spent on developing the actual functionality of the software. To make COCOMO suitable in the changed scenario, Boehm proposed COCOMO 2 [Boehm95] in 1995.

COCOMO 2 provides three models to arrive at increasingly accurate cost estimations. These can be used to estimate project costs at different phases of the software product. As the project progresses, these models can be applied at the different stages of the same project.

Application composition model: This model as the name suggests, can be used to estimate the cost for prototype development. We had already discussed in Chapter 2 that a prototype is usually developed to resolve user interface issues.

Early design model: This supports estimation of cost at the architectural design stage.

Post-architecture model: This provides cost estimation during detailed design and coding stages.

The post-architectural model can be considered as an update of the original COCOMO. The other two models help consider the following two factors. Now a days every software is interactive and GUI-driven. GUI development constitutes a significant part of the overall development effort. The second factor concerns several issues that affect productivity such as the extent of reuse. We briefly discuss these three models in the following.

Application composition model

The application composition model is based on counting the number of screens, reports, and modules (components). Each of these components is considered to be an object (this has nothing to do with the concept of

objects in the object-oriented paradigm). These are used to compute the object points of the application.

Effort is estimated in the application composition model as follows:

1. Estimate the number of screens, reports, and modules (components) from an analysis of the SRS document.
2. Determine the complexity level of each screen and report, and rate these as either simple, medium, or difficult. The complexity of a screen or a report is determined by the number of tables and views it contains.
3. Use the weight values in Table 3.3 to 3.5.

The weights have been designed to correspond to the amount of effort required to implement an instance of an object at the assigned complexity class.

Table 3.3: SCREEN Complexity Assignments for the Data Tables

Number of views	Tables < 4	Tables < 8	Tables ≥ 8
< 3	Simple	Simple	Medium
3–7	Simple	Medium	Difficult
>8	Medium	Difficult	Difficult

Table 3.4: Report Complexity Assignments for the Data Tables

Number of views	Tables < 4	Tables < 8	Tables ≥ 8
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 or more	Medium	Difficult	Difficult

4. Add all the assigned complexity values for the object instances together to obtain the object points.

Table 3.5: Table of Complexity Weights for Each Class for Each Object Type

Object type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component	—	—	10

5. Estimate percentage of reuse expected in the system. Note that reuse refers to the amount of pre-developed software that will be used within the system. Then, evaluate New Object-Point count (NOP) as follows,

$$NOP = \frac{(\text{Object-Points})(100 - \%\text{ of reuse})}{100}$$

6. Determine the productivity using Table 3.6. The productivi ty depends onthe experience of the developers as well as the maturity of the CASE environment used.
7. Finally, the estimated effort in person-months is computed as E = NOP/PROD.

Table 3.6: Productivity Table

Developers' experience	Very low	Low	Nominal	High	Very high
CASE maturity	Very low	Low	Nominal	High	Very high
PRODUCTIVITY	4	7	13	25	50

Early design model

T he unadjusted function points (UFP) are counted and converted to source lines of code (SLOP). In a typical programming environment, each UFP would correspond to about 128 lines of C, 29 lines of C++, or 320 lines of assembly code. Of course, the conversion from UFP to LOC is environment specific, and depends on factors such as extent of reusable libraries supported. Seven cost drivers that characterise the post-architecture model are used. These are rated on a seven points scale. The cost drivers include product reliability and complexity, the extent of reuse, platform sophistication, personnel experience, CASE support, and schedule.

The effort is calculated using the following formula:

$$\text{Effort} = \text{K SLOC} \times \square_i \text{ cost driver}_i$$

Post-architecture model

The effort is calculated using the following formula, which is similar to the original COCOMO model.

$$\text{Effort} = a \times \text{K SLOC}^b \times \square_i \text{ cost driver}_i$$

The post-architecture model differs from the original COCOMO model in the choice of the set of cost drivers and the range of values of the exponent b. The exponent b can take values in the range of 1.01 to 1.26. The details of the COCOMO 2 model, and the exact values of b and the cost drivers can be found in [Boehm 97].

3.7 HALSTEAD'S SOFTWARE SCIENCE—AN ANALYTICAL TECHNIQUE

Halstead's software science[2] is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum volume, actual volume, language level, effort, and development time.

For a given program, let:

- h_1 be the number of unique operators used in the program,
- h_2 be the number of unique operands used in the program,
- N_1 be the total number of operators used in the program,
- N_2 be the total number of operands used in the program.

Although the terms operators a nd operands have intuitive meanings, a precise definition of these terms is needed to avoid ambiguities. But, unfortunately we would not be able to provide a precise definition of these two terms. There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages. However, a few general guidelines regarding identification of operators and operands for any programming language can be provided. For instance, assignment, arithmetic, and logical operators are usually counted as operators.

A pair of parentheses, as well as a block begin —block end pair, are considered as single operators. A label is considered to be an operator, if it is used as the target of a GOTO statement. The constructs `if ... then ... else ... endif` and `a while ... do` are considered as single operators. A sequence (statement termination) operator ';' is considered as a single operator. Subroutine declarations and variable declarations comprise the operands. Function name in a function call statement is considered as an operator, and the arguments of the function call are considered as operands. However, the parameter list of a function in the function declaration statement is not considered as operands. We list below what we consider to be the set of operators and operands for the ANSI C language. However, it should be realised that there is considerable disagreement among various researchers in this regard.

Operators and Operands for the ANSI C language

The following is a suggested list of operators for the ANSI C language:

```
( [ . , -> * + - ~ ! ++ -- * / % + - << >> < > <= >= !=
== & ^ | && || = *= /= %= += -= <<= >>= &= ^= |= : ? { ;

CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE
BREAK RETURN and a function name in a function call
```

Operands are those variables and constants which are being used with operators in expressions. Note that variable names appearing in declarations are not considered as operands.

Example 3.3 Consider the expression a = &b; a, b are the operands and =, & are the operators.

Example 3.4 The function name in a function definition is not counted as an operator.

```
int func ( int a, int b )
{
      . . .
}
```

For the above example code, the operators are: {}, ( ) We do not consider func, a, and b as operands, since these are part of the function definition.

Example 3.5 Consider the function call statement: func (a, b);. In this, func ‘ ,’ a n d ; are considered as operators and variables a, b are treated as operands.

3.7.1 Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length N = N_1 + N_2. Halstead’s definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notion of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary $h = h_1 + h_2$.

3.7.2 Program Volume

The length of a program (i.e., the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used.

Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 h$$

Let us try to understand the important idea behind this expression. Intuitively, the program volume V is the minimum number of bits needed to encode the program. In fact, to represent h different identifiers uniquely, we need at least $\log_2 h$ bits (where h is the program vocabulary). In this scheme, we need $N \log_2 h$ bits to store a program of length N. Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

3.7.3 Potential Minimum Volume

The potential minimum volume V* is defined as the volume of the most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction, say a function call like `foo()`;. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands. Note that the operands are the input and output data items.

Thus, if an algorithm operates on input and output data d_1, d_2, ... d_n, the most succinct program would be $f(d_1, d_2, ..., d_n)$; for which, $h_1 = 2$, $h_2 = n$. Therefore, $V^* = (2 + h_2) \log_2 (2 + h_2)$.

The program level L is given by $L = V^*/V$. The concept of program level L has been introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in

assembly language than to develop a program in a high-level language to solve a problem.

3.7.4 Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort E = V /L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort $E = V^2/V^*$ (since $L = V^*/V$) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer's time T = E/S, where S is the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

3.7.5 Length Estimation

Even though the length of a program can be found by calculating the tota l number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc., can be determined even before the start of any programming activity. His method is summarised below.

Halstead assumed that it is quite unlikely that a program has several identical parts— in formal language terminology identical substrings—of length greater than h(h being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is usually made into a procedure or a function. Thus, we can safely assume that any program of length N consists of N/h unique strings of length h. Now, it is a standard combinatorial result that for any given alphabet of size K, there are exactly K^r different strings of length r. Thus,

$$\frac{N}{\eta} \leq \eta^{\eta}$$

or

$$N \leq \eta^{\eta+1}$$

Since operators and operands usually alternate in a program, we can

further refine the upper bound into N ≤ $hh_1^{h1}\ h_2^{h3}$. Also, N must include not only the ordered set of N elements, but it should also include all possible subsets of that ordered set, i.e. the power set of N strings

(This particular reasoning of Halstead is hard to justify!).

Therefore,

$$2^N = \eta\eta_1^{\eta_1}\eta_2^{\eta_2}$$

or, taking logarithm on both sides,

$$N = \log_2 \eta + log_2(\eta_1^{\eta_1}\eta_2^{\eta_2})$$

So, we get,

$$N = \log_2(\eta_1^{\eta_1}\eta_2^{\eta_2}) \quad \text{(approximately, by ignoring } \log_2 \eta)$$

or,

$$N = log_2\eta_1^{\eta_1} + log_2\eta_2^{\eta_2}$$
$$= \eta_1 log_2\eta_1 + \eta_2 log_2\eta_2$$

Experimental evidence gathered from the analysis of a large number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs are considered individually.

Example 3.6 Let us consider the following C program:

```
main()
{
  int a,b,c,avg;
  scanf("%d %d %d",&a,&b,&c);
  avg=(a+b+c)/3;
  printf("avg= %d",avg);
}
```

The unique operators are: `main, (), {}, int, scanf, &, “,”, “;”, =, +, /, printf`

The unique operands are: `a,b,c,&a,&b,&c,a+b+c,avg,3,”%d %d %d”, “avg=%d”`

Therefore,

$$\eta_1 = 12, \eta_2 = 11$$
$$\text{Estimated Length} = (12 * \log 12 + 11 * \log 11)$$
$$= (12 * 3.58 + 11 * 3.45) = (43 + 38) = 81$$
$$\text{Volume} = \text{Length} * \log(23) = 81 * 4.52 = 366$$

In conclusion, Halstead's theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead's software science provides gross estimates of properties of a large collection of software, but extends to individual cases rather inaccurately.

3.8 RISK MANAGEMENT

Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go hay ware.

> A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway.

We need to distinguish between a risk which is a problem that might occur from the problems currently being faced by a project. If a risk becomes real, the anticipated problem becomes a reality and is no more a risk. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project. Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared beforehand to contain each risk. In this context, risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real. Risk management consists of three essential activities—risk identification, risk assessment, and risk mitigation. We discuss these three activities in the following subsections.

3.8.1 Risk Identification

The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks is minimised. So, early risk identification is important. Risk identification is somewhat similar to the project manager listing down his nightmares. For example, project manager might be worried whether the vendors whom you have asked to develop certain modules might not complete their work in time, whether they would turn in poor quality work, whether some of your key personnel might leave the organisation, etc. All such

risks that are likely to affect a project must be identified and listed.

A project can be subject to a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project: project risks, technical risks, and business risks. We discuss these risks in the following.

Project risks: Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus he can accurately assess the progress of the work and control it, if he finds any activity is progressing at a slower rate than what was planned. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

Technical risks: Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due the development team's insufficient knowledge about the product.

Business risks: This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

Classification of risks in a project

Example 3.12 Let us consider a satellite based mobile communication product discussed in Case Study 2.2 of Section 2.5. The project manager can identify several risks in this project. Let us classify them appropriately.

- What if the project cost escalates and overshoots what was estimated?: **Project risk.**
- What if the mobile phones that are developed become too bulky in size

to conveniently carry?: **Business risk.**

- What if it is later found out that the level of radiation coming from the phones is harmful to human being?: **Business risk.**
- What if call hand-off between satellites becomes too difficult to implement?: **Technical risk.**

In order to be able to successfully foresee and identify different risks that might affect a software project, it is a good idea to have a company disaster list. This list would contain all the bad events that have happened to software projects of the company over the years including events that can be laid at the customer's doors. This list can be read by the project mangers in order to be aware of some of the risks that a project might be susceptible to. Such a disaster list has been found to help in performing better risk analysis.

3.8.2 Risk Assessment

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

- The likelihood of a risk becoming real (r).
- The consequence of the problems associated with that risk (s).

Based on these two factors, the priority of each risk can be computed as follows:

$$p = r \times s$$

where, p is the priority with which the risk must be handled, r is the probability of the risk becoming real, and s is the severity of damage caused due to the risk becoming real. If all identified risks are prioritised, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for those risks.

3.8.3 Risk Mitigation

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures. In fact, most risks require considerable ingenuity on the part of the project manager in tackling the risks.

There are three main strategies for risk containment:

Avoid the risk: Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are:

Process-related risk: These risks arise due to aggressive work schedule, budget, and resource utilisation.

Product-related risks: These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability etc.

Technology-related risks: These risks arise due to commitment to use certain technology (e.g., satellite communication).

A few examples of risk avoidance can be the following: Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.

Transfer the risk: This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use. For example, if you are using a compiler for recognising user commands, you would have to construct a compiler for a small and very primitive command language first.

There can be several strategies to cope up with a risk. To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this we may compute the risk leverage of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

$$\text{risk leverage} = \frac{\text{risk exposure before reduction} - \text{risk exposure after reduction}}{\text{cost of reduction}}$$

Even though we identified three broad ways to handle any risk, effective risk handling cannot be achieved by mechanically following a set procedure,

but requires a lot of ingenuity on the part of the project manager. As an example, let us consider the options available to contain an important type of risk that occurs in many software projects—that of schedule slippage.

REQUIREMENTS ANALYSIS AND SPECIFICATION

4.1 REQUIREMENTS GATHERING AND ANALYSIS

The complete set of requirements are almost never available in the form of a single document from the customer. In fact, it would be unrealistic to expect the customers to produce a comprehensive document containing a precise description of what he wants. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources.

We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

- Requirements gathering
- Requirements analysis

We discuss these two tasks in the following subsections.

4.1.1 Requirements Gathering

Requirements gathering is also popularly known as *requirements elicitation*.

The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.

Requirements gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually. In this case, a working model of the system (that is, the manual system) exists. Availability of a working model is usually of great help in requirements gathering.

Typically even before visiting the customer site, requirements gathering

activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site, the analysts normally interview the end-users and customer representatives,[1] carry out requirements gathering activities such as

In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1. Studying existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the developers. Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

2. Interview: Typically, there are many differe nt categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants. The library members would like to use the software to query availability of books and issue and return books. The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc. The accounts personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.

To systematise this method of requirements gathering, the Delphi technique can be followed. In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This procedure is repeated till the different users agree on the set of requirements.

3. Task analysis: The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to

formulate the different steps necessary to realise the required functionality in consultation with the users. For example, for the issue book service, the steps may be—authenticate user, check the number of books issued to the customer and determine if the maximum number of books that this member can borrow has been reached, check whether the book has been reserved, post the book issue details in the member's record, and finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.

Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different types of scenarios of a task, the behaviour of the software can be different. For example, the possible scenarios for the book issue task of a library automation software may be:

- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.
- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users. For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.

Form analysis: Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to b e filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis the exiting forms and the formats of the notifications produced are analysed to determine the data input to the system and the data that are output from the system. For the different sets of data input to the system, how these input data would be used by the system to produce the corresponding output data is determined from the users.

4.1.2 Requirements Analysis

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness, since each stakeholder typically has only a partial and incomplete view of the software. Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.

The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

For carrying out requirements analysis effectively, the analyst first needs to develop a clear grasp of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

- What is the problem?
- Why is it important to solve the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the possible procedures that need to be followed to solve the problem?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what should be the data interchange formats with the external systems?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.

During requirements analysis,the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Let us examine these different types of requirements problems in detail.

Anomaly: It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development. The following are two examples of anomalous requirements:

problems existing in the gathered requirements?

Many of the inconsistencies, anomalies, and incompleteness are detected effortlessly, while some others require a focused study of the specific requirements. A few problems in the requirements can, however, be very subtle and escape even the most experienced eyes. Many of these subtle anomalies and inconsistencies can be detected, if the requirements are specified and analysed using a formal method. Once a system has been formally specified, it can be systematically (and even automatically) analysed to remove all problems from the specification. We will discuss the basic concepts of formal system specification in Section 4.3. Though the use of formal techniques is not widespread, the current practice is to formally specify only the safety-critical parts of a system.

4.2 SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise the requirements in the form of an SRS document. The SRS document usually contains all the user requirements in a structured though an informal form.

Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write. One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience. In the following subsection, we discuss the different categories of users of an SRS document and their needs from it.

4.2.1 Users of SRS Document

Usually a large number of different people need the SRS document for

very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

Users, customers, and marketing personnel: These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.

Software developers: The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.

Test engineers: The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.

User documentation writers: The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.

Pro ject managers: The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.

Maintenance engineers: The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code. Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

Many software engineers in a project consider the SRS document to be a reference document. However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer. In fact, the SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future. The SRS document can even be used as a legal document to settle disputes between the customers and the developers in a court of law. Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the

requirements mentioned in the SRS document.

4.2.2 Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work. In the following subsection, we identify the important uses of a well-formulated SRS document:

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The pr oject manager also uses the SRS document for work scheduling.

Provides a baseline for validation and verification: The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements.

Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

4.2.3 Characteristics of a Good SRS Document

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. IEEE Recommended Practice

for Software Requirements Specifications[IEEE830] describes the content and qualities of a good software requirements specification (SRS). Some of the identified desirable qualities of an SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behaviour of the system a nd not discuss the implementation issues.

Traceable: It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyse the impact of changing a requirement on the design elements and the code.

Modifiable: Customers frequently change the requirements during the software development development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify. Having the description of a requirement scattered across many places in the SRS document may not be wrong—but it tends to make the requirement difficult to understand and also any modification to the requirement would become difficult as it would require changes to be made at large number of places in the document.

Identification of response to undesired events: The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation. A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement—"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

4.2.4 Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. As discussed earlier, the most damaging problems are incompleteness, ambiguity, and contradictions. There are many other types problems that a specification document might suffer from. By knowing these problems, one can try to avoid them while writing an SRS document. Some of the important categories of problems that many SRS documents suffer from are as follows:

Over-specification: It occurs when the analyst tries to address the "how to" aspects in the SRS document. For example, in the library automation problem, one should not specify whether the library membership records need to be stored indexed on the member's first name or on the library member's identification (ID) number. Over-specification restricts the freedom of the designers in arriving at a good design solution.

Forward references: One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.

Wishful thinking: This type of problems concern description of aspects which would be difficult to implement.

Noise: The term noise refers to presence of material not directly relevant to the software development process. For example, in the register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8am and 5pm, 7 days a week. This information can be called *noise* as it would hardly be of any use to the software developers and would unnecessarily clutter the SRS document,

diverting the attention from the crucial points.

Several other "sins" of SRS documents can be listed and used to guard against writing a bad SRS document and is also used as a checklist to review an SRS document.

4.2.5 Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following.

An SRS document should clearly document the following aspects of a software:
- Functional requirements
- Non-functional requirements
  - — Design and implementation constraints
  - — External interfaces required
  - — Other non-functional requirements
- Goals of implementation.

In the following subsections, we briefly describe the different categories of requirements.

Functional requirements

The functional requirements capture the functionalities required by the users from the system.

The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set. documented effectively.

Non-functional requirements

The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.

In the following subsections, we discuss the different categories of non-functional requirements that are described under three different sections:

Design and implementation constraints: Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers. Some of the example constraints can be—corporate or regulatory policies that needs to be honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.

External interfaces required: Examples of external interfaces are—hardware, software and communication interfaces, user interfaces, report formats, etc. To specify the user interfaces, each interface between the software and the users must be described. The description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.

Other non-functional requirements: This section contains a description of non- functional requirements that are neither design constraints and nor are external interface requirements. An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in this section may include reliability issues, accuracy of results, and security issues.

Goals of implementation

The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed. These are not binding on the developers, and they may take these suggestions into account if possible. For example, the developers may use these suggestions while

choosing among different design solutions.

The goals of implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately. It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

4.2.6 Functional Requirements

In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements. The high-level functions would be split into smaller subrequirements. Each high-level function is an instance of use of the system (use case) by the user in some way.

A high-level function is one using which the user can get some useful piece of work done.

In Figure 4.2, the different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behaviour exhibited by the system for the same high-level function. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement can consist of several sub-requirements.

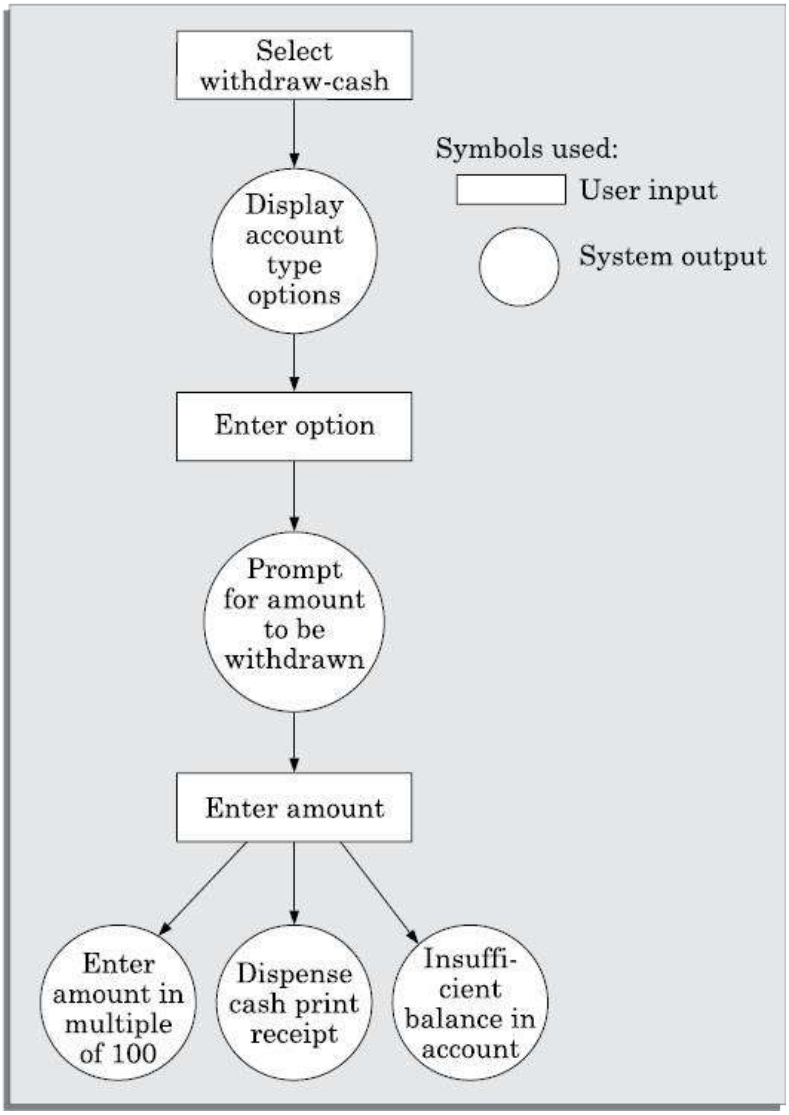


Figure 4.2: User and system interactions in high-level functional requirement.

4.2.7 How to Identify the Functional Requirements?

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem.

Remember that there can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to first identify the different types of users who might use the system and then try to identify the different services expected from the software by different types of users.

The decision regarding which functionality of the system can be taken to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some

subjectivity. For example, consider the issue-book function in a Library Automation System. Suppose, when a user invokes the issue-book function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the issue-book function? Many times, the choice is obvious. But, sometimes it requires making non-trivial decisions.

4.2.8 How to Document the Functional Requirements?

Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented. A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. We now illustrate the specification of the functional requirements through two examples. Let us first try to document the withdraw-cash function of an *automated tell e r machine* (ATM) system in the following. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These user interaction sequences may vary from one invocation from another depending on some conditions. These different interaction sequences capture the different *scenarios*. To accurately describe a functional requirement, we must document all the different scenarios that may occur.

4.2.9 Traceability

Traceability means that it would be possible to identify (trace) the specific design component which implements a given requirement, the code part that corresponds to a given design component, and test cases that test a given requirement. Thus, any given code component can be traced to the corresponding design component, and a design component can be traced to a specific requirement that it implements a nd *vice versa*. Traceability analysis is an important concept and is frequently used during software development. For example, by doing a traceability analysis, we can tell whether all the requirements have been satisfactorily addressed in all phases. It can also be used to assess the impact of a requirements change. That is, traceability makes it easy to identify which parts of the design and code would be affected, when certain requirement change occurs. It can also be used to study the impact of a bug that is known to exist in a code part on various

requirements, etc.

4.2.10 Organisation of the SRS Document

I n this section, we discuss the organisation of an SRS document as prescribed by the IEEE 830 standard[IEEE 830]. Please note that IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it, as may be required for specific projects. Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst. However, organisation of the SRS document to a large extent depends on the preferences of the system analyst himself, and he is often guided in this by the policies and standards being followed by the development company. Also, the organisation of the document and the issues discussed in it to a large extent depend on the type of the product being developed. However, irrespective of the company's principles and product type, the three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.

The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics. The introduction section may include the hardware that the system will run on, the devices that the system will interact with and the user skill-levels. Description of the user skill-level is important, since the command language design and the presentation styles of the various documents depend to a large extent on the types of the users it is targeted for. For example, if the skill-levels of the users is "novice", it would mean that the user interface has to be very simple and rugged, whereas if the user-level is "advanced", several short cut techniques and advanced features may be provided in the user interface.

It is desirable to describe the formats for the input commands, input data, output reports, and if necessary the modes of interaction. We have already

discussed how the contents of the Sections on the functional requirements, the non-functional requirements, and the goals of implementation should be written. In the following subsections, we outline the important sections that an SRS document should contain as suggested by the IEEE 830 standard, for each section of the document, we also briefly discuss the aspects that should be discussed in it.

4.3 FORMAL SYSTEM SPECIFICATION

In recent years, formal techniques[3] have emerged as a central issue in software engineering. This is not accidental; the importance of precise specification, modelling, and verification is recognised to be important in most engineering disciplines. Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. We say a system is correctly implemented when it satisfies its given specification. The specification of a system can be given either as a list of its desirable properties (property-oriented approach) or as an abstract model of the system (model-oriented approach). These two approaches are discussed here. Before discussing representative examples of these two types of formal specification techniques, we first discuss a few basic concepts in formal specification We will first highlight some important concepts in formal methods, and examine the merits and demerits of using formal techniques.

4.3.1 What is a Formal Technique?

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by its specification language. More precisely, a formal specification language consists of two sets—syn and sem, and a relation sat between them. The set *syn* is called the *syntactic domain*, the set *sem* is called the *semantic domain*, and the relation *sat* is called the *satisfaction relation*. For a given specification *syn*, and model of the system *sem*, if *sat (syn, sem)*, then *syn* is said to be the *specification of sem*, and *sem* is said to be the *specificand* of *syn*.

T he generally accepted paradigm for system development is through a hierarchy of abstractions. Each stage in this hierarchy is an implementation of

its preceding stage and a specification of the succeeding stage. The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.

Syntactic domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

Semantic domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronisation trees, partial orders, state machines, etc.

Satisfaction relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined— those that *preserve* a system's behaviour and those that *preserve* a system's structure.

Model *versus* property-oriented methods

Formal methods are usually classified into two broad categories—the so-called *model-oriented* and the property-oriented approaches. In a *model-*

oriented style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the *property-oriented* style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Let us consider a simple producer/consumer example. In a *property-oriented* style, we would probably start by listing the properties of the system like—the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. Two examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In *a model-oriented* style, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that S 1 + p ⇒ S, S + c ⇒ S 1. Thus model-oriented approaches essentially specify a program by writing another, presumably simpler program. A few notable examples of popular model-oriented specification techniques are Z, CSP,CCS, etc.

It is alleged that property-oriented approaches are more suitable for *requirements specification*, and that the model-oriented approaches are more suited to *system design specification*. The reason for this distinction is the fact that property-oriented approaches specify a system behaviour not by what they say of the system but by what they do not say of the system. Thus, property-oriented specifications permit a large number of possible implementations.

4.3.2 Operational Semantics

Informally, the *operational semantics* of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a *single run* of the system and how the runs are grouped together to describe the *behaviour* of the system. In the following subsection we discuss some of the commonly used operational semantics.

Linear semantics: In this approach, a *run* o f a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the atomic actions. For example, a concurrent activity a || b is represented by the set of sequential activities a; b a nd b; a. This is a simple but rather unnatural

representation of concurrency. The behaviour of a system in this model consists of the set of all its runs. To make this model more realistic, usually *justice* and *fairness* restrictions are imposed on computations to exclude the unwanted interleavings.

Branching semantics: In this approach, the behaviour of a system is represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

Maximally parallel semantics: In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial order semantics: Under this view, the semantics ascribed to a system is a *structure of states* satisfying a partial order relation among the states (events). The partial order represents a *precedence ordering* among events, and constrains some events to occur only after some other events have occurred; while the occurrence of other events (called *concurrent events*) is considered to be incomparable. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

Merits and limitations of formal methods

In addition to facilitating precise formulation of specifications, formal methods possess several positive features, some of which are discussed as follows:

- Formal specifications encourage rigour. It is often the case that the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification. It is widely acknowledged that it is cost-effective to spend more efforts at the specification stage, otherwise, many flaws would go unnoticed only to be detected at the later stages of software development that would lead to iterative changes to occur in the development life cycle. According to an estimate, for large and complex systems like distributed real-time

systems 80 per cent of project costs and most of the cost overruns result from the iterative changes required in a system development process due to inappropriate formulation of requirements specification. Thus, the additional effort required to construct a rigorous specification is well worth the trouble.

- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications. Informal specifications may be useful in understanding a system and its documentation, but they cannot serve as a basis of verification. Even carefully written specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to unverified programs. automatically avoided when one formally specifies a system.
- The mathematical basis of the formal methods makes it possible for automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback o n the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a "toy" working model of a system that can provide immediate feedback on the behaviour of the specified system, and is especially useful in checking the completeness of specifications.

It is clear that formal methods provide mathematically sound frameworks within which large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal meth ods suffer from several shortcomings, some of which are as following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check *absolute* correctness of systems using theorem proving techniques.

- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

In the following two sections, we discuss the axiomatic and algebraic specification styles. Both these techniques can be classified as the property-oriented specification techniques.

4.4 AXIOMATIC SPECIFICATION

In axiomatic specification of a system, first-order logic is used to write the pre- and post- conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

How to develop an axiomatic specifications?

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Establish the constraints on the input parameters as a predicate.
- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type assertion is not necessary for pure functions.
- Combine all of the above into pre- and post-conditions of the function.

We now illustrate how simple abstract data types can be algebraically specified through two simple examples.

4.5 ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980,1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Essentially, algebraic specifications define a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g. { I, +, -, *, / }. In contrast, alphabetic strings S together with operations of concatenation and length {S, I , con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in a heterogeneous algebra is called a *sort* of the algebra. To define a heterogeneous algebra, besides defining the sorts, we need to specify the involved operations, their signatures, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using *equations*. An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the *signature* of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

Equations section: This section gives a set of *rewrite rules* (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

By convention each equation is implicitly universally quantified over all possible values of the variables. This means that the equation holds for all possible values of the variable. Names not mentioned in the syntax section

such r or e are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

Basic construction operators: These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, 'create' and 'append' are basic construction operators in Example 4.13.

Extra construction operators: These are the construction operators other than the basic construction operators. For example, the operator 'remove' in Example 4.13 is an extra construction operator, because e ven without using 'remove' it is possible to generate all values of the type being specified.

Basic inspection operators: These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified—these are the inspection operators. The set of the basic operators S1 is a subset of S , such that each operator from S -S 1 can be expressed in terms of the operators from S 1.

Extra inspection operators: These are the inspection operators that are not basic inspectors. A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in Example 4.13, create is a constructor because point appears on the right hand side of the expression and point is the data type being specified. But, xcoord is an inspection operator since it does not modify the point type.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

Completeness: This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. When the equations are not complete, at some step during the reduction process, we might not be able to reduce the expression arrived at that step by using any of the equations. There is no simple procedure to ensure that an algebraic specification is complete.

Finite termination property: This property essentially addresses the

following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable.

But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.

Unique termination property: This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked—Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique termination property is a very difficult problem.

4.5.1 Structured Specification

Developing algebraic specifications is time consuming. Therefore efforts have been made to devise ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

Incremental specification: The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.

Specification instantiation: This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Pros and Cons of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to integrate with typical programming languages. Also, algebraic specifications are hard to understand.

SOFTWARE DESIGN

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. We can state the main objectives of the design phase, in other words, as follows.

This view of a design process has been shown schematically in Figure 5.1. As shown in Figure 5.1, the design process starts using the SRS document and completes with the production of the design document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

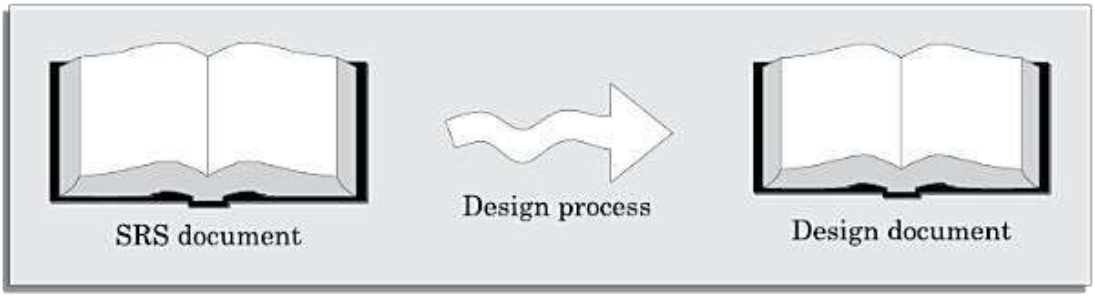


Figure 5.1: The design process.

5.1 OVERVIEW OF THE DESIGN PROCESS

The design process essentially transforms the SRS document into a design document. In the following sections and subsections, we will discuss a few important issues associated with the design process.

5.1.1 Outcome of the Design Process

The following items are designed and documented during the design phase.

Different modules required: The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and

data necessary to accomplish the task of registration of the students should be named handle student registration.

Control relationships among modules: A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules: The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules: Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required to implement the individual modules: Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps that we are going to discuss in this chapter and the subsequent three chapters. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

5.1.2 Classification of Design Activities

A good software design is seldom realised by using a single step procedure, rather it requires iterating over a series of steps called the design activities. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.

- Preliminary (or high-level) design, and
- Detailed design.

The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level

design as follows:

The outcome of high-level design is called the program structure or the software architecture. High-level design is a crucial step in the overall design of a software. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy. Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the structure chart. Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.

5.1.3 Classification of Design Methodologies

The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms. In this chapter, we shall discuss the important characteristics of these two fundamental design approaches. Over the next three chapters, we shall study these two approaches in detail.

Do design techniques result in unique solutions?

Even while using the same design methodology, different designers usually arrive at very different design solutions. The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives. As a result, it is possible that even the same designer can work out many different solutions to the same problem. Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one. However, a fundamental question that arises at this point is—how to distinguish superior design solution from an inferior one? Unless we know what a good software design is and how to distinguish a superior design solution from an inferior one, we can not possibly design one. We investigate this issue in the next section.

5.2 HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Coming up with an accurate characterisation of a good software design that would hold across diverse problem domains is certainly not easy. In fact, the definition of a "good" software design can vary depending on the exact application being designed. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. These characteristics are listed below:

Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design solution should adequately address resource, time, and cost optimisation issues.

Maintainability: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

5.2.1 Understandability of a Design: A Ma jor Concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one?

Recollect from our discussions in Chapter 1 that a good design should help overcome the human cognitive limitations that arise due to limited short-term memory. A large problem overwhelms the human mind, and a poor design would make the matter worse. Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs. Therefore, a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain. A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it.

An understandable design is modular and layered

How can the understandability of two different designs be compared, so that we can pick the better one? To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess. A design solution should have the following characteristics to be easily understandable:

- It should assign consistent and meaningful names to various design components.
- It should make use of the principles of decomposition and abstraction in good measures to simplify the design.

We had discussed the essential concepts behind the principles of abstraction and decomposition principles in Chapter 1. But, how can the abstraction and decomposition principles are used in arriving at a design solution? These two principles are exploited by design methodologies to make a design modular and layered. (Though there are also a few other forms in which the abstraction and decomposition principles can be used in the design solution, we discuss those later). We can now define the characteristics of an easily understandable design as follows: A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

A design solution should be modular and layered to be understandable.

We now elaborate the concepts of modularity and layering of modules:

Modularity

A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other. Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly. To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually.

A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

A software design with high cohesion and low coupling among modules is the effective problem decomposition we discussed in Chapter 1. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

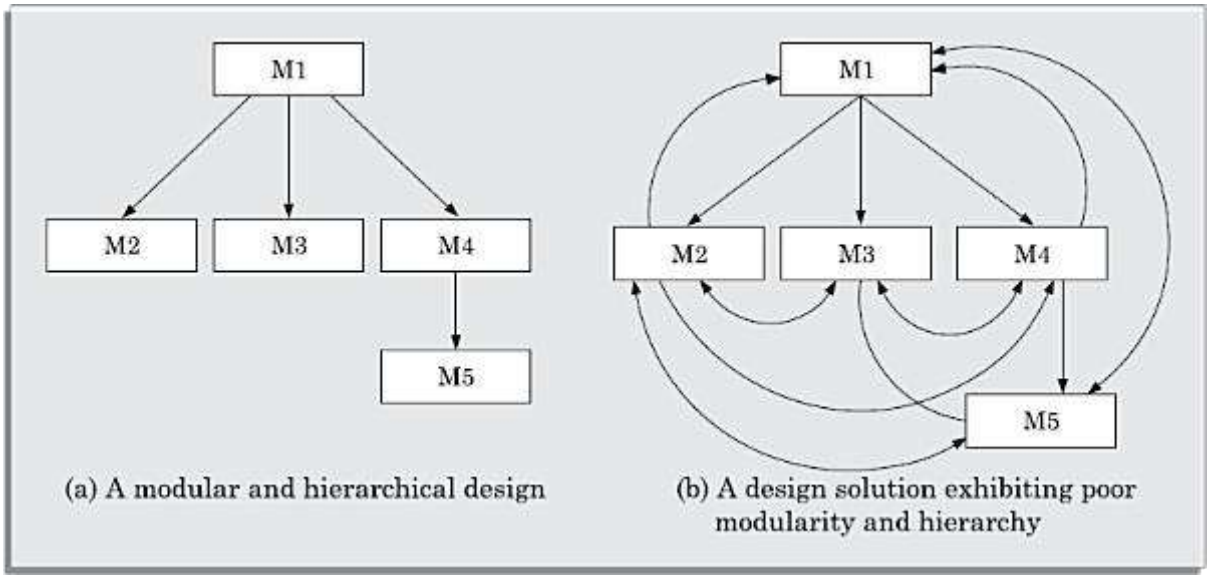


Figure 5.2: Two design solutions to the same problem.

Based on this classification, we would be able to easily judge the cohesion and coupling existing in a design solution. From a knowledge of the cohesion and coupling in a design, we can form our own opinion about the modularity of the design solution. We shall define the concepts of cohesion and coupling and the various classes of cohesion and coupling in Section 5.3. Let us now discuss the other important characteristic of a good design solution—layered design.

Layered design

A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done. A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.

A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to

understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error. We shall elaborate these concepts governing layered design of modules in Section 5.4.

5.3 COHESION AND COUPLING

We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other. Let us now define what is meant by cohesion and coupling.

Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

In this section, we first elaborate the concepts of cohesion and coupling. Subsequently, we discuss the classification of cohesion and coupling.

Coupling: Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:

- If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.

If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion: To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution. When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor

cohesion.

Functional independence

By the term functional independence, we mean that a module performs a single task and needs very little interaction with other modules.

> A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.

Functional independence is a key to any good design primarily due to the following advantages it offers:

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it. This is especially so, if the module accesses the data (or code) internal to other modules.

Understandability: When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other. We have already pointed out in Section 5.2 that understandability is a major advantage of a modular design. Besides the three we have listed here, there are many other advantages of a modular design as well. We shall not list those here, and leave it as an assignment to the reader to identify them.

5.3.1 Classification of Cohesiveness

Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom. However, the different classes of cohesion that modules can possess are depicted in Figure 5.3. The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

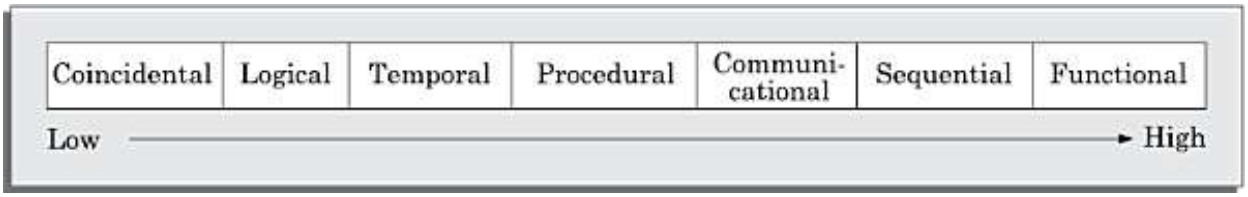


Figure 5.3: Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily. An example of a module with coincidental cohesionhas been shown in Figure 5.4(a).Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

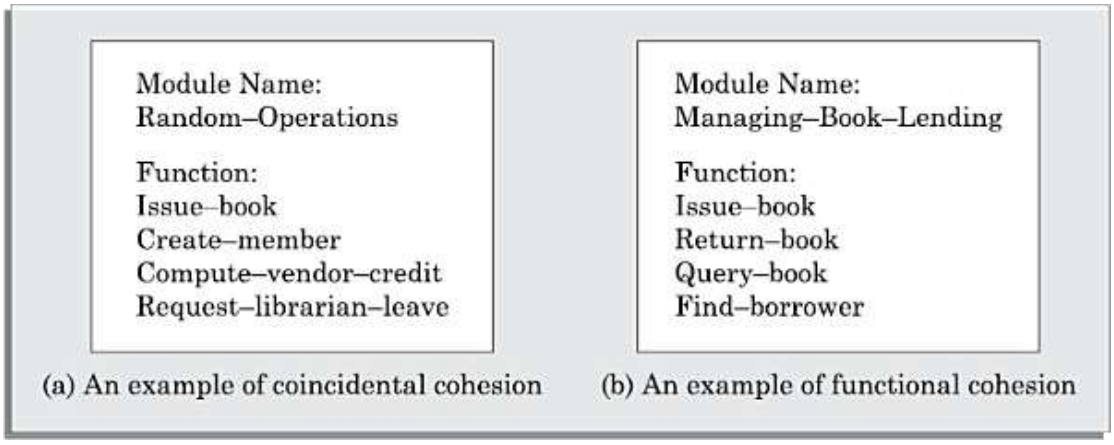


Figure 5.4: Examples of cohesion.

Logical cohesion: A module is said to be logically cohesive, if all

elements of the module perform similar operations, such as error handling, data input, data output, etc. As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialisation of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion. Other examples of modules having temporal cohesion are the following. Similarly, a module would exhibit temporal cohesion, if it comprises functions for performing initialisation, or start-up, or shut-down of some process.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print-bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

Sequential cohesion: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence. As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), place-order-on-vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order() creates an order that is processed by the function check-item-availability() (whether

the items are available in the required quantities in the inventory) is input to place-order-on-vendor().

Functional cohesion: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees. Another example of a module possessing functional cohesion has been shown in Figure 5.4(b). In this example, the functions issue-book(), return-book(), query-book(), and find-borrower(), together manage all activities concerned with book lending. When a module possesses functional cohesion, then we should be able to describe what the module does using only one simple sentence. For example, for the module of Figure 5.4(a), we can describe the overall responsibility of the module by saying "It manages the book lending procedure of the library."

5.3.2 Classification of Coupling

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.

The degree of coupling between two modules depends on their interface complexity.

The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.

Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities have also been shown in Figure 5.5.

Figure 5.5: Classification of coupling.

Data coupling: Two modules are data coupled, if they communicate using

an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

The different types of coupling are shown schematically in Figure 5.5. The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

5.4 APPROACHES TO SOFTWARE DESIGN

There are two fundamentally different approaches to software design that are in use today— function-oriented design, and object-oriented design. Though these two design approaches are radically different, they are complementary rather than competing techniques. The object-oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following. Salient features of these two approaches are discussed in subsections 5.5.1 and 5.5.2 respectively.

5.4.1 Function-oriented Design

The following are the salient features of the function-oriented design approach:

Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.

In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

For example, consider a function create-new-library m e m be r which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This high-level function may be refined into the following subfunctions:

- assign-membership-number
- create-member-record
- print-bill

Each of these subfunctions may be split into more detailed subfunctions and so on.

Centralised system state: The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event. For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules.

The system state is centralised and shared among different functions.

For example, in the library management system, several functions such as the following share data such as member-records for reference and updation:

- create-new-member
- delete-member
- update-member-record

A large number of function-oriented design approaches have been proposed in the past. A few of the well-established function-oriented design approaches are as following:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]

- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

5.4.2 Object-oriented Design

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities). Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is decentralised since there is no globally shared data in the system and data is stored in each object. For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs). The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language introduced in the 1970s. ADT is an important concept that forms an important pillar of object-orientation. Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—data abstraction, data structure, data type. We discuss these in the following subsection:

Data abstraction: The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organised, and manipulated inside the object. The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.

Data structure: A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organised collection of primitive data items such as integer, floating point numbers, characters, etc.

Data type: A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.

In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs? Let us examine the three main advantages of using ADTs in programs:

- The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localises the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.
- An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular.
- Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus an object may exist in different states depending the values of the internal data. In different states, an object may behave differently. We shall elaborate these concepts in Chapter 7 and subsequently we discuss an object-oriented design methodology in Chapter 8.

Object-oriented versus function-oriented design approaches

The following are some of the important differences between the

function-oriented and object-oriented design:

- Unlike function-oriented design methods in OOD, the basic abstraction is not the services available to the users of the system such as issue-book, display-book-details, find-issued-books, etc., but real-world entities such as member, book, book-register, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc., but by designing objects such as employees, departments, etc.
- In OOD, state information exists in the form of data distributed among several objects of the system. In contrast, in a procedural design, the state information is available in a centralised shared data store. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc., are usually implemented as global data in a traditional programming system; whereas in an object-oriented design, these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by sending a message to it. Of course, somewhere or other the real-world functions must be implemented.
- Function-oriented techniques group functions together if, as a group, they constitute a higher level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function-oriented design approaches, let us consider an example—that of an automated fire-alarm system for a large building.

Automated fire-alarm system—customer requirements

The owner of a large multi-storied building wants to have a computerised fire alarm system designed, developed, and installed in his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire has been sensed and then sound the alarms

only in the neighbouring locations. The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel would man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Function-oriented approach: In this approach, the different high-level functions are first identified, and then the data structures are designed.

```
/* Global data (system state) accessible by various functions */
        BOOL  detector_status[MAX_ROOMS];
        int   detector_locs[MAX_ROOMS];
        BOOL  alarm-status[MAX_ROOMS]; /* alarm activated when status is set */
        int   alarm_locs[MAX_ROOMS]; /* room number where alarm is located */
        int   neighbour-alarms[MAX-ROOMS][10]; /* each detector has at most */
                                               /* 10 neighbouring alarm locations */
        int   sprinkler[MAX_ROOMS];
```

The functions which operate on the system state are:

```
interrogate_detectors();
get_detector_location();
determine_neighbour_alarm();
determine_neighbour_sprinkler();
ring_alarm();
activate_sprinkler();
reset_alarm();
reset_sprinkler();
report_fire_location();
```

Object-oriented approach: In the object-oriented approach, the different classes of objects are identified. Subsequently, the methods and data for each object are identified. Finally, an appropriate number of instances of each class is created.

```
class detector
attributes: status, location, neighbours
operations: create, sense-status, get-location,
       find-neighbours

class alarm
attributes: location, status
operations: create, ring-alarm, get_location, reset-
alarm

class sprinkler
```

```
attributes: location, status
operations: create, activate-sprinkler, get_location,
reset-sprinkler
```

We can now compare the function-oriented and the object-oriented approaches based on the two examples discussed above, and easily observe the following main differences:

- In a function-oriented program, the system state (data) is centralised and several functions access and modify this central data. In case of an object-oriented program, the state information (data) is distributed among various objects.
- In the object-oriented design, data is private in different objects and these are not available to the other objects for direct access and modification.
- The basic unit of designing an object-oriented program is objects, whereas it is functions and modules in procedural designing. Objects appear as nouns in the problem description; whereas functions appear as verbs.

FUNCTION-ORIENTED SOFTWARE DESIGN

6.1 OVERVIEW OF SA/SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

- During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

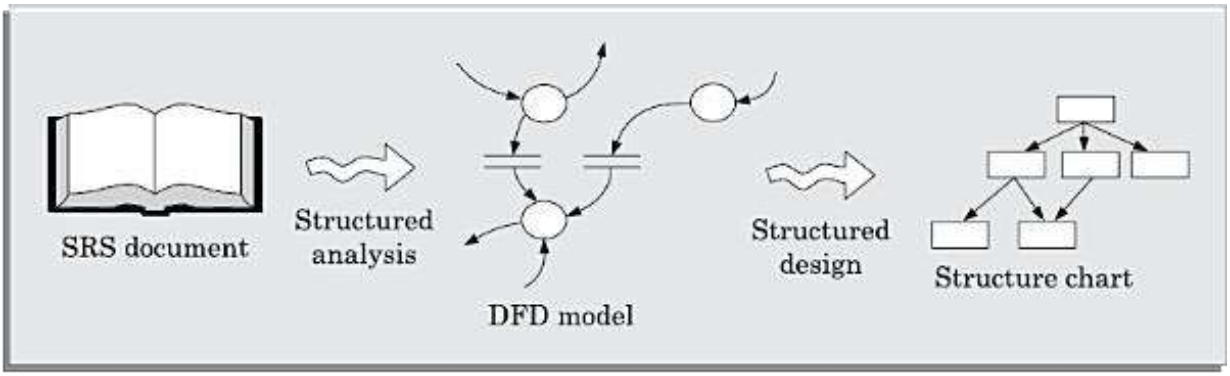


Figure 6.1: Structured analysis and structured design methodology.

As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.

The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

> It is important to understand that the purpose of structured analysis is to capture the detailed structure of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for implementation in some programming language.

The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user's terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.

In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

6.2 STRUCTURED ANALYSIS

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using data flow diagrams (DFDs).

DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.

Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used

by the functions, etc. In the DFD terminology, each function is called a process or a bubble. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.

DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organisation.

6.2.1 Data Flow Diagrams (DFDs)

The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system. The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams. We had pointed out while discussing the principle of abstraction in Section 1.3.2 that any hierarchical representation is an effective means to tackle complexity. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:

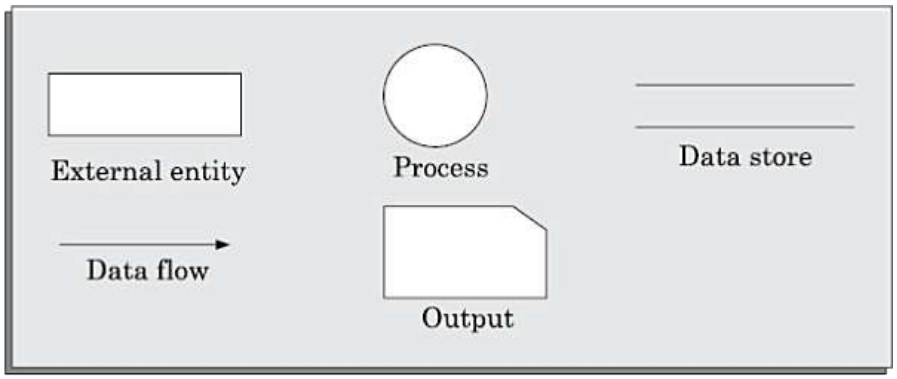


Figure 6.2: Symbols used for designing DFDs.

Function symbol: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).

External entity symbol: An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.

Data flow symbol: A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read-number to validate-number, data-item flowing into read-number, and valid-number flowing out of validate-number.

Data store symbol: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting t o a data store need not be annotated with the name of the corresponding data items. As an example of a data store, number is a data store in Figure 6.3(b).

Output symbol: The output symbol i s as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

The notations that we are following in this text are closer to the Yourdon's notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

Important concepts associated with constructing DFD models

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at t he same speed. An example of such an arrangement is shown in Figure 6.3(a). Here, the `validate-number` bubble can start processing only after t h e read-number bubble has supplied data to it; and the `read-number` bubble has to wait until the `validate-number` bubble has consumed its data.

However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

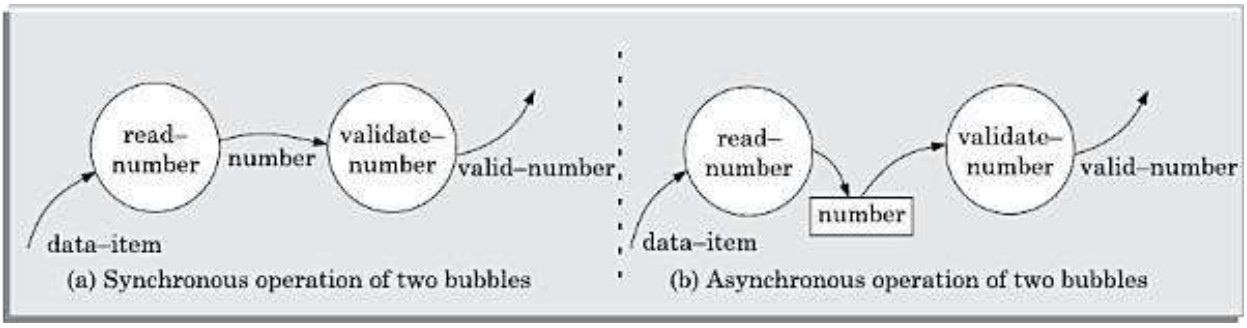


Figure 6.3: Synchronous and asynchronous data flow.

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary. A

data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc., as shown in Figure 6.4 discussed in new subsection. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

$$grossPay = regularPay + overtimePay$$

For the smallest units of data items, the data dictionary simply lists their name and their type. Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.

The dictionary plays a very important role in any software development process, especially for the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.

- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

For large systems, the data dictionary can become extremely complex and voluminous. Even moderate-sized projects can have thousands of entries in

the data dictionary. It becomes extremely di fficult to maintain a voluminous dictionary manually. Computer-aided software engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary. As a result, the designers do not have to spend almost any effort in creating the data dictionary. These CASE tools also support some query language facility to query about the definition and usage of data items. For example, queries may be formulated to determine which data item affects which processes, or a process affects which data items, or the definition and usage of specific data items, etc. Query handling is facilitated by storing the data dictionary in a relational database management system (RDBMS).

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

+: denotes composition of two data items, e.g. a+b represents data a and b.

[,,]: represents selection, i.e. any one of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs or b occurs.

(): the contents inside the bracket represent optional data which may or may not appear.

a+(b) represents either a or a+b occurs.

{}: represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

=: represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.

/* */: Anything appearing within /* and */ is considered as comment.

6.3 DEVELOPING THE DFD MODEL OF A SYSTEM

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.

The DFD model of a problem consists of many of DFDs and a single data dictionary.

The DFD model of a system i s constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system

(highest level). It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

6.3.1 Context Diagram

The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble. The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality. As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as 'Supermarket software'.

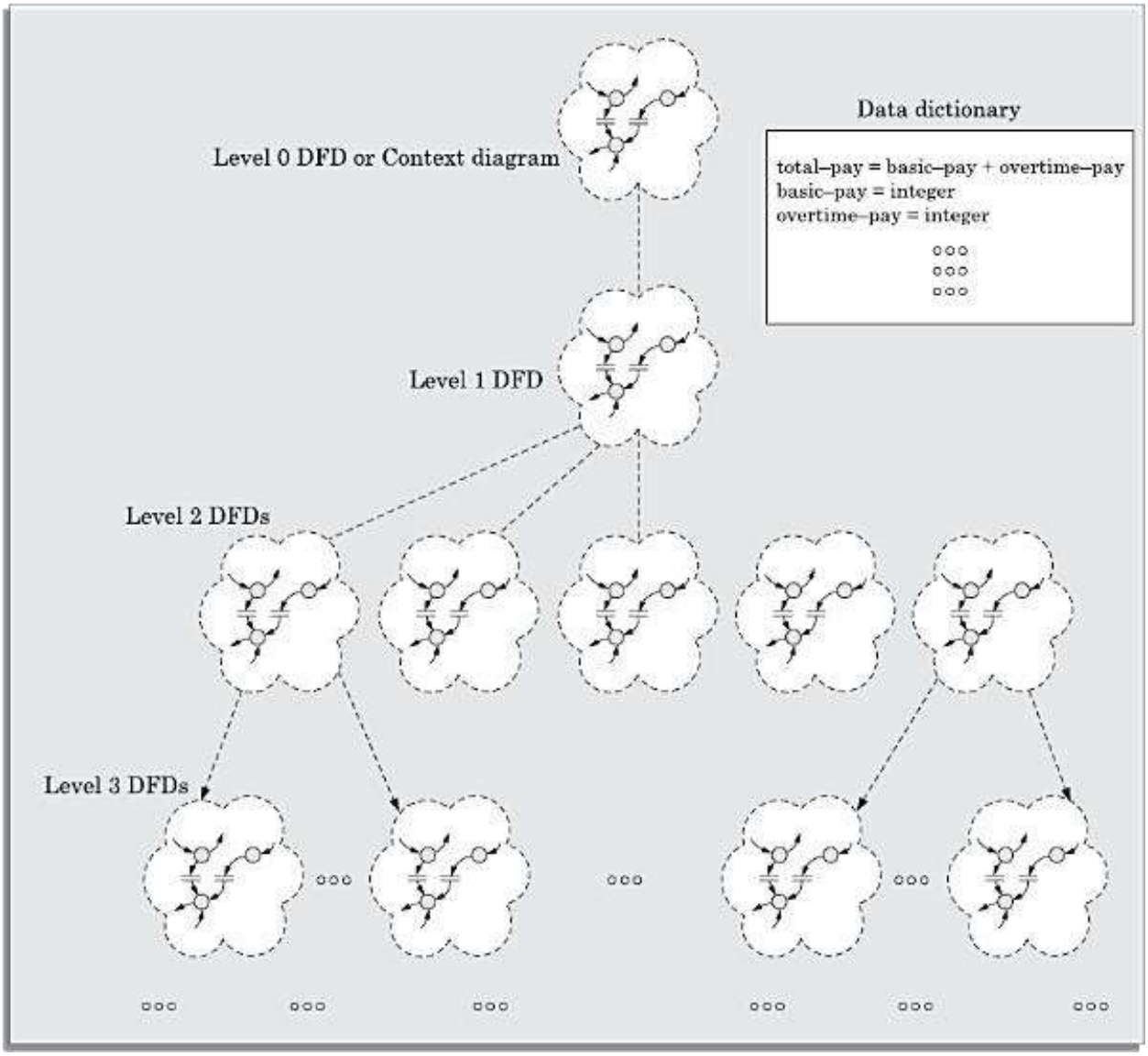


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

The context diagram establishes the context in which the system operates; that is, who are the users, what data do they input to the system, and what data they received by the system.

The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data

names.

To develop the context diagram of the system, we have to analyse the SRS document to identify the different types o f users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also includes any external systems which supply data to or receive data from the system.

6.3.2 Level 1 DFD

The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high-level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

What if a system has more than seven high-level requirements identified in the SRS document? In this case, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD. These can be split appropriately in the lower DFD levels. If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their subfunctions so that we have roughly about five to seven bubbles represented on the diagram. We illustrate construction of level 1 DFDs in Examples 6.1 to 6.4.

Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring o r exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level m a k e that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles (i.e. more than seven bubbles) at any level o f a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried

on until a level is reached at which the function of the bubble can be described using a simple algorithm.

We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:
   - Different high-level functions that the system needs to perform.
   - Data input to every high-level function.
   - Data output from every high-level function.
   - Interactions (data flow) among the identified high-level functions.

   Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.

   Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

   Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:
   - Identify the different subfunctions of the high-level function.
   - Identify the data input to each of these subfunctions.
   - Identify the data output from each of these subfunctions.
   - Identify the interactions (data flow) among these subfunctions.

   Represent these aspects in a diagrammatic form using a DFD.

   Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is

decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing DFDs

The DFD model of a system usually consists of many DFDs that are organised in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD.

> The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.

We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1,0.1.2,0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d 2 flows in. Please note that dangling arrows (d1,d2,d3) represent the data flows into or out of a diagram.

How far to decompose?

A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions. For simple problems, decomposition up to level 1 should suffice. However, large industry standard problems may need decomposition up to level 3 or level 4. Rarely, if ever, decomposition beyond level 4 is needed.

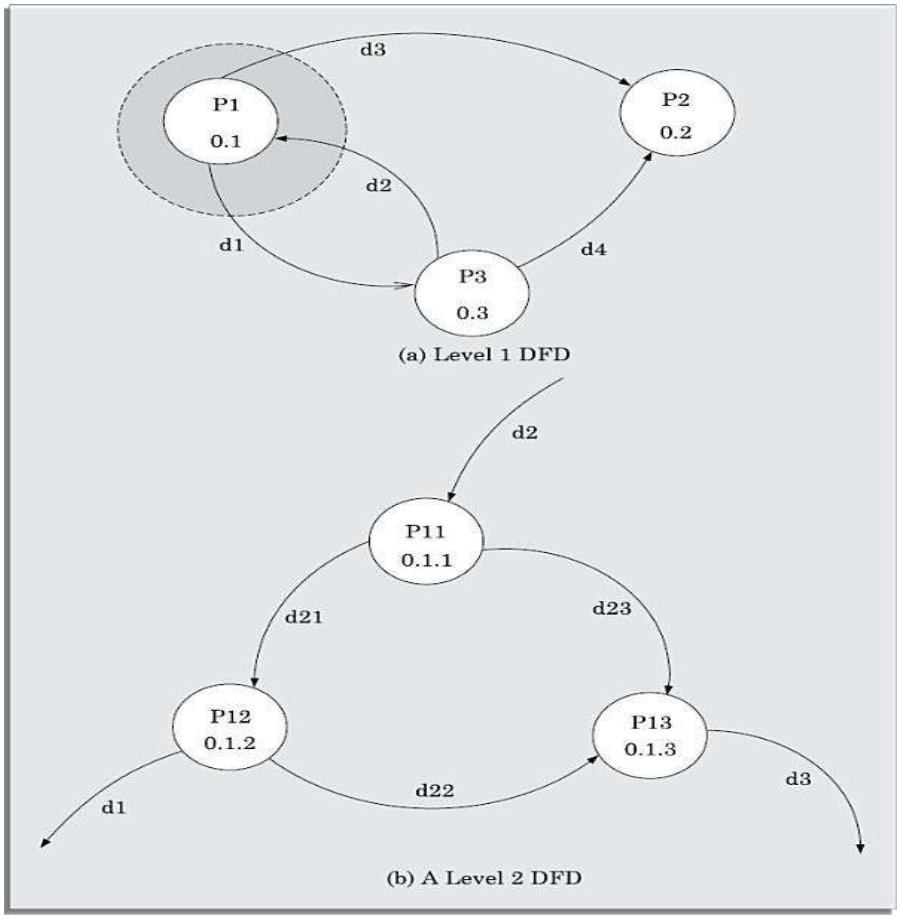


Figure 6.5: An example showing balanced decomposition.

Commonly made errors while constructing a DFD model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore useful to understand the different types of mistakes that beginners usually make while constructing the DFD model

of systems, so that you can consciously try to avoid them.The errors are as follows:

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.

- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

It is important to realise that a DFD represents only data flow, and it does not represent any control information.

The following are some illustrative mistakes of trying to represent control aspects such as:

Illustration 1. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While developing the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in Figure 6.6) to indicate that the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

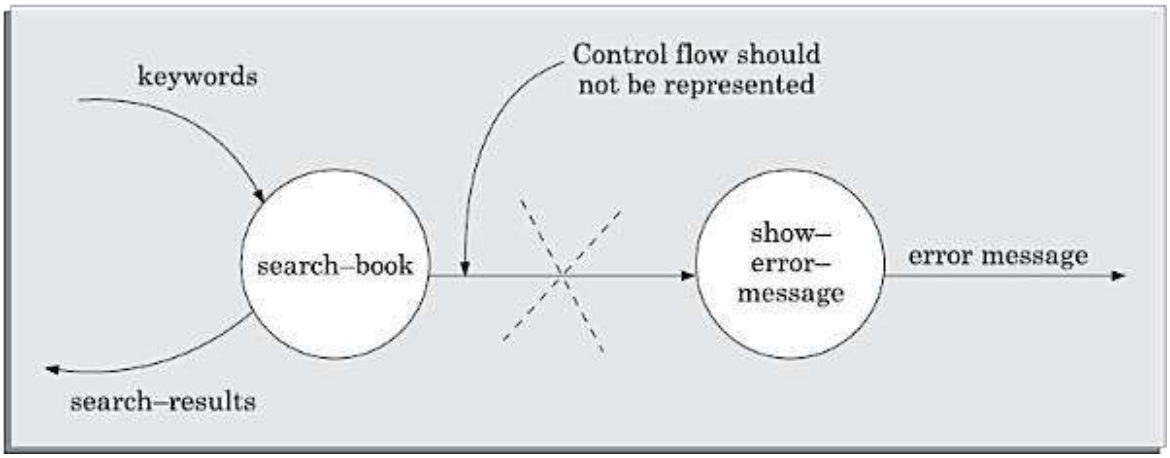


Figure 6.6: It is incorrect to show control information on a DFD.

Illustration 2. Another type of error occurs when one tries to represent when or in what order different functions (processes) are invoked. A DFD similarly should not represent the conditions under which different functions are invoked.

Illustration 3. If a bubble A invokes either the bubble B or the bubble C

depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data flow arrow should not connect two data stores or even a data store with an external entity. Thus, data cannot flow from a data store to another data store or to an external entity without any intervening processing. As a result, a data store should be connected only to bubbles through data flow arrows.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in the SRS document of the system should be overlooked.
- Only those functions of the system specified in the SRS document should be represented. That is, the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
- Incomplete data dictionary and data dictionary showing incorrect composition of data items are other frequently committed mistakes.
- The data and function names must be intuitive. Some students and even practicing developers use meaningless symbolic data names such as a,b,c, etc. Such names hinder understanding the DFD model.
- Novices usually clutter their DFDs with too many data flow arrow. It becomes difficult to understand a DFD if any bubble is associated with more than seven data flows. When there are too many data flowing in or out of a DFD, it is better to combine these data items into a high-level data item. Figure 6.7 shows an example concerning how a DFD can be simplified by combining several data flows into a single high-level data flow.

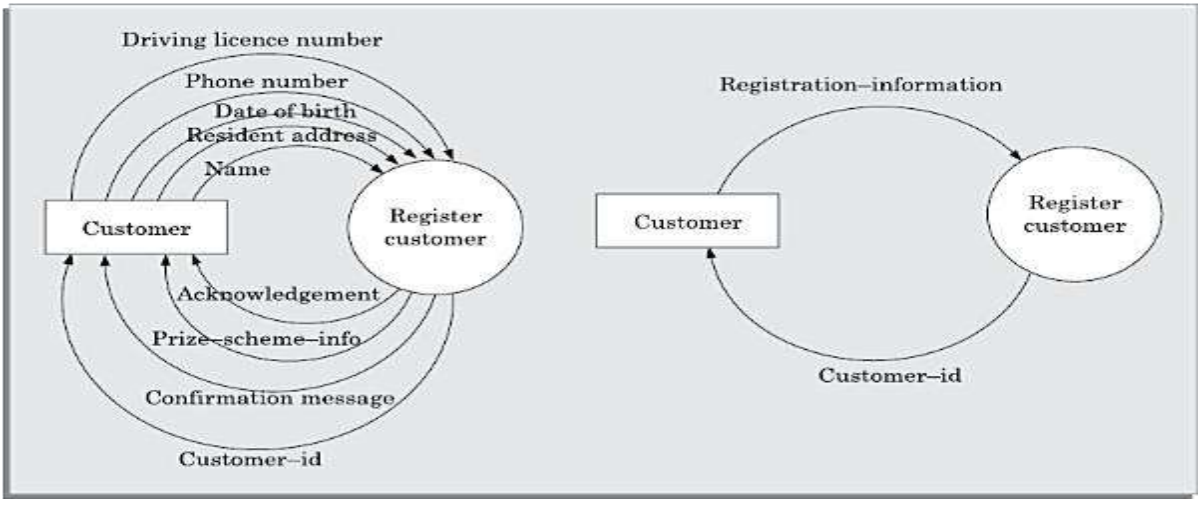


Figure 6.7: Illustration of how to avoid data cluttering.

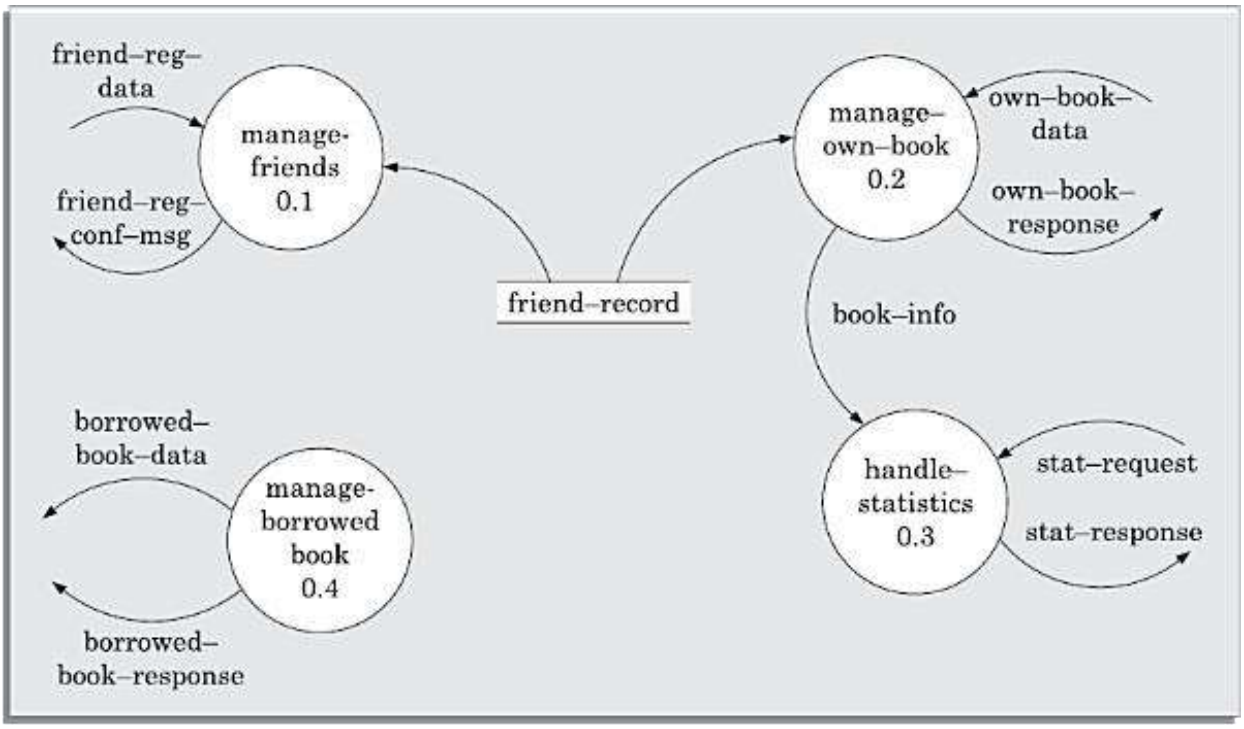


Figure 6.16: Level 1 DFD for Example 6.5.

The level 2 DFD for the manageOwnBook bubble is shown in Figure 6.17.

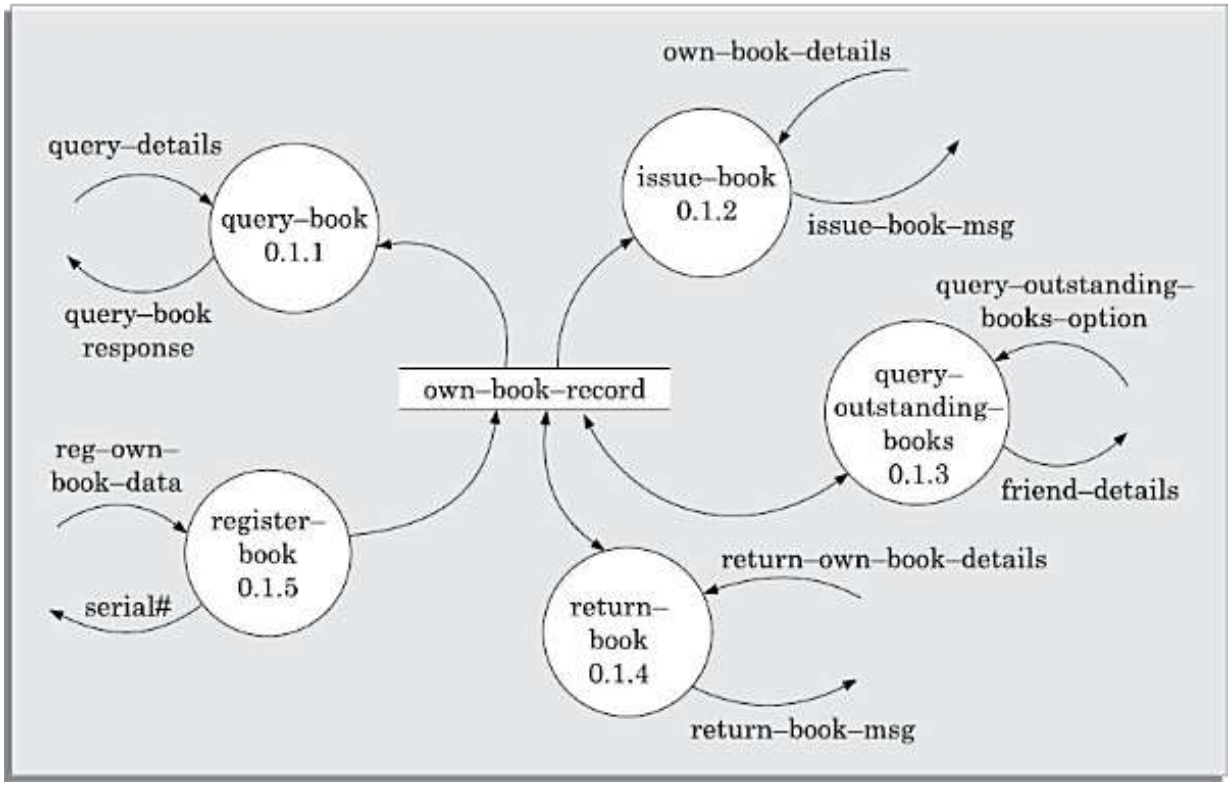


Figure 6.17: Level 2 DFD for Example 6.5.

6.3.3 Extending DFD Technique to Make it Applicable to Real-time Systems

In a real-time system, some of the high-level functions are associated with deadlines. Therefore, a function must not only produce correct results but also should produce them by some prespecified time. For real-time systems, execution time is an important consideration for arriving at a correct design. Therefore, explicit representation of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a control flow diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

- The effect of an external event or control signal.
- The processes that are invoked as a consequence of an event.

Control specifications represents the behavior of the system in two different ways:

- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinatorial specification of behaviour. PAT represents invocation sequence of bubbles in a DFD.

6.4 STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure

chart represents the software architecture. The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks using which structure charts are designed are as following:

Rectangular boxes: A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows: An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a modules calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flo w arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.

Library modules: A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.

Selection: The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.

Repetition: A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

In any structure chart, there should be one and only one module at the top, called the root. There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A. The main reason behind this

restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules. However, it is possible for t wo higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

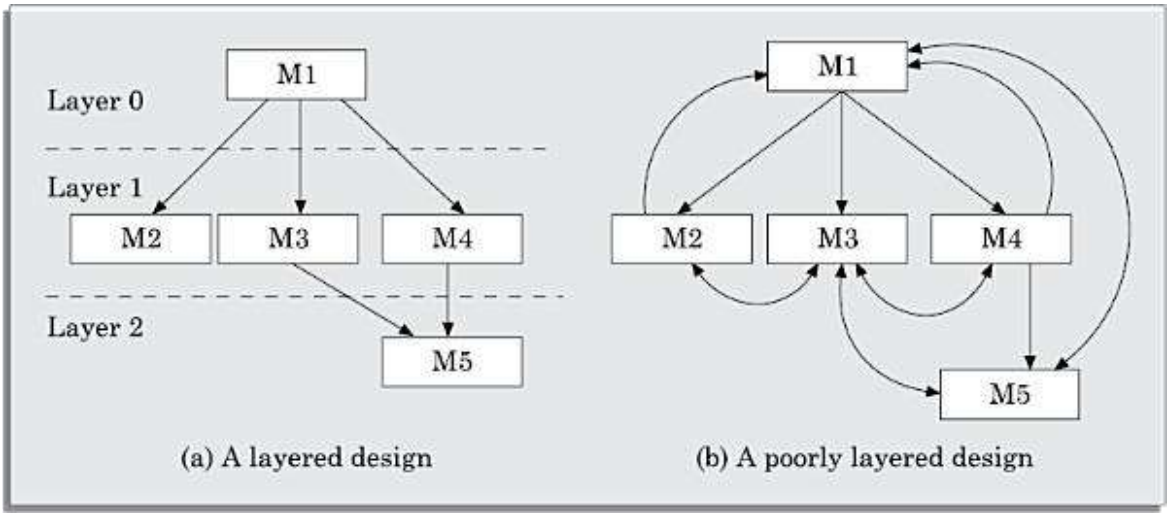


Figure 6.18: Examples of properly and poorly layered designs.

Flow chart ***versus*** structure chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flo w of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that i s inherent to a flow chart is suppressed in a structure chart.

6.4.1 Transformation of a DFD Model into Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Transform analysis

Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:

- Input.
- Processing.
- Output.

The input portion in the DFD includes processes that transform input data from physical (e.g, character from terminal) to logical form (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the input and output parts requires experience and skill. One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms. Processes which sort input or filter data from it are central tansforms. T h e first level o f structure chart is produced by representing each input and output unit as a box and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialisation and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

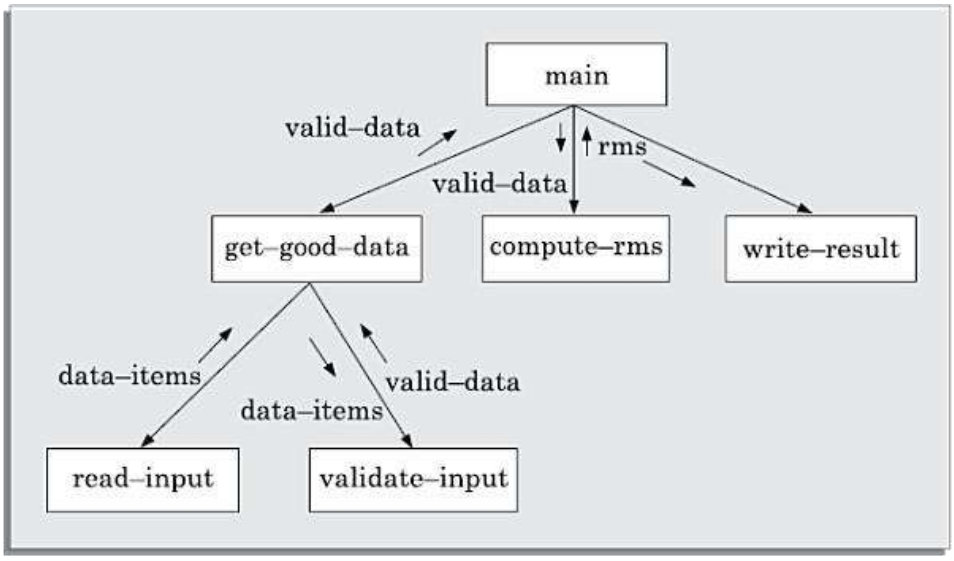


Figure 6.19: Structure chart for Example 6.6.

Transaction analysis

Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.

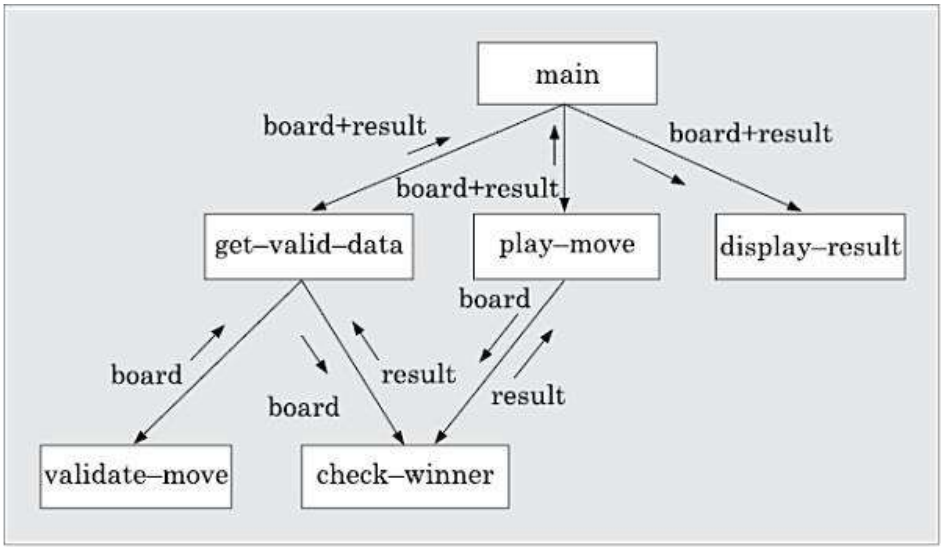


Figure 6.20: Structure chart for Example 6.7.

As in transform analysis, first all data entering into the DFD need to be identified. In a transaction-driven system, different data items may pass through different computation paths through the DFD. This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps. Each different way in which input data is processed is a transaction. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modules a nd a transaction-center module.

6.5 DETAILED DESIGN

During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules. The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

6.6 DESIGN REVIEW

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, members of the team

who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:

Traceability: Whether each bubble of the DFD can be traced to some module in the structure chart a nd vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.

Correctness: Whether all the algorithms and data structures of the detailed design are correct.

Maintainability: Whether the design can be easily maintained in future.

Implementation: Whether the design can be easily and efficiently be implemented.

After the points raised by the reviewers is addressed by the designers, the design document becomes ready for implementation.

CODING AND TESTING

10.1 CODING

The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design. The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified. During the coding phase, different modules identified in the design document are coded according to their respective module specifications. We can describe the overall objective of the coding phase to be the following.

> The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.

Normally, good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard. These software development organisations formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously because of the significant business advantages it offers. The main advantages of adhering to a standard style of coding are the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It facilitates code understanding and code reuse.
- It promotes good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, the error return conventions, etc. Besides the coding standards, several coding guidelines are also prescribed by software companies. But, what is the difference between a coding guideline and a coding standard?

After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. It is important to detect as many errors as possible

during code reviews, because reviews are an efficient way of removing errors from code as compared to defect elimination using testing. We first discuss a few representative coding standards and guidelines.

10.2 CODE REVIEW

Testing is an effective defect removal mechanism. However, testing is applicable to only executable code. Review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing. Over the years, review techniques have become extremely popular and have been generalised for use with other work products.

Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Obviously, code review does not target to design syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors. Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.

The reason behind why code review is a much more cost-effective strategy to eliminate errors from code compared to testing is that reviews directly detect errors. On the other hand, testing only helps detect failures and significant effort is needed to locate the error during debugging.

The rationale behind the above statement is explained as follows. Eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time consuming activity. In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error.

Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss

these two code review techniques.

10.2.1 Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).

The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.

Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. These guidelines are based on personal experience, common sense, several other subjective factors. Therefore, these guidelines should be considered as examples rather than as accepted rules to be applied dogmatically. Some of these guidelines are following:

- The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
- In order to foster co-operation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

10.2.2 Code Inspection

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs. We can state the principal aim of the code inspection to be the following:

The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed to another programmer's errors.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more lik ely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.

Good software development companies collect statistics regarding different types of errors that are commonly committed by their engineers and identify the types of errors most frequently committed. Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialised variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatch between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values.
- Dangling reference caused when the referenced memory has not been allocated.

10.2.3 Clean Room Testing

Clean room testing was pioneered at IBM. This type of testing relies

heavily on walkthroughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. It is interesting to note that the term cleanroom was first coined at IBM by drawing analogy to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing. The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors. Also testing- based error detection is efficient for detecting certain errors that escape manual inspection.

10.3 SOFTWARE DOCUMENTATION

When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are helpful in the following ways:

- Good documents help enhance understandability of code. As a result, the availability of good documents help to reduce the effort and time required for maintenance.
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover[1] problem. Even when an engineer leaves the organisation, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
- Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.

Different types of software documents can broadly be classified into the following:

Internal documentation: These are provided in the source code itself.

External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.

We discuss these two types of documentation in the next section.

10.3.1 Internal Documentation

Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following:

- Comments embedded in the source code.
- Use of meaningful variable names.
- Module and function headers.
- Code indentation.
- Code structuring (i.e., code decomposed into modules and functions).
- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

Out of these different types of internal documentation, which one is the most valuable for understanding a piece of code?

The above assertion, of course, is in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without much thought. For example, the following style of code commenting is not much of a help in understanding the code.

```
a=10; /* a made 10 */
```

A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments. Good software development organisations usually ensure good internal documentation by appropriately formulating their coding standards

and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

10.3.2 External Documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc. A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.

An important feature that is requierd of any good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

Gunning's fog index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group

o f words that can be independently pronounced. For example, the word "sentence" has three syllables ("sen", "ten", and "ce"). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

10.4 TESTING

The aim of program testing is to help realize identify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Consider a function taking a floating point number as argument. If a tester takes 1sec to type in a value, then even a million testers would not be able to exhaustively test it after trying for a million number of years. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

In this section, we will discuss a few basic concepts in program testing on which our subsequent discussions on program testing would be based.

How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction. A highly simplified view of program testing is schematically shown in Figure 10.1. The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs. Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers. For examples, a software might fail for a test case only

when a network connection is enabled.

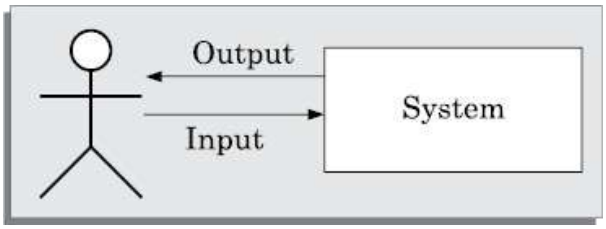


Figure 10.1: A simplified view of program testing.

Terminologies

As is true for any specialised domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.
- An **error** is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function.

Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community. Please note that in the domain of hardware testing, the term fault is used with a slightly different connotation [IEEE90] as compared to the terms error and bug.

Verification versus validation

The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different. We summarise the main differences between these two techniques in the following:

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- The primary techniques used for verification include review, simulation, formal verification, and testing. Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker. On the other hand, validation techniques are primarily based on product testing. Note that we have categorised testing both under program verification and validation. The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is a s per the module and module interface specifications. On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
- Verification does not require execution of the software, whereas

validation requires execution of the software.

- Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

10.4.2 Testing Activities

Testing involves performing the following main activities:

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques. We discuss a few important test case design techniques later in this Chapter.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analysed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected.Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

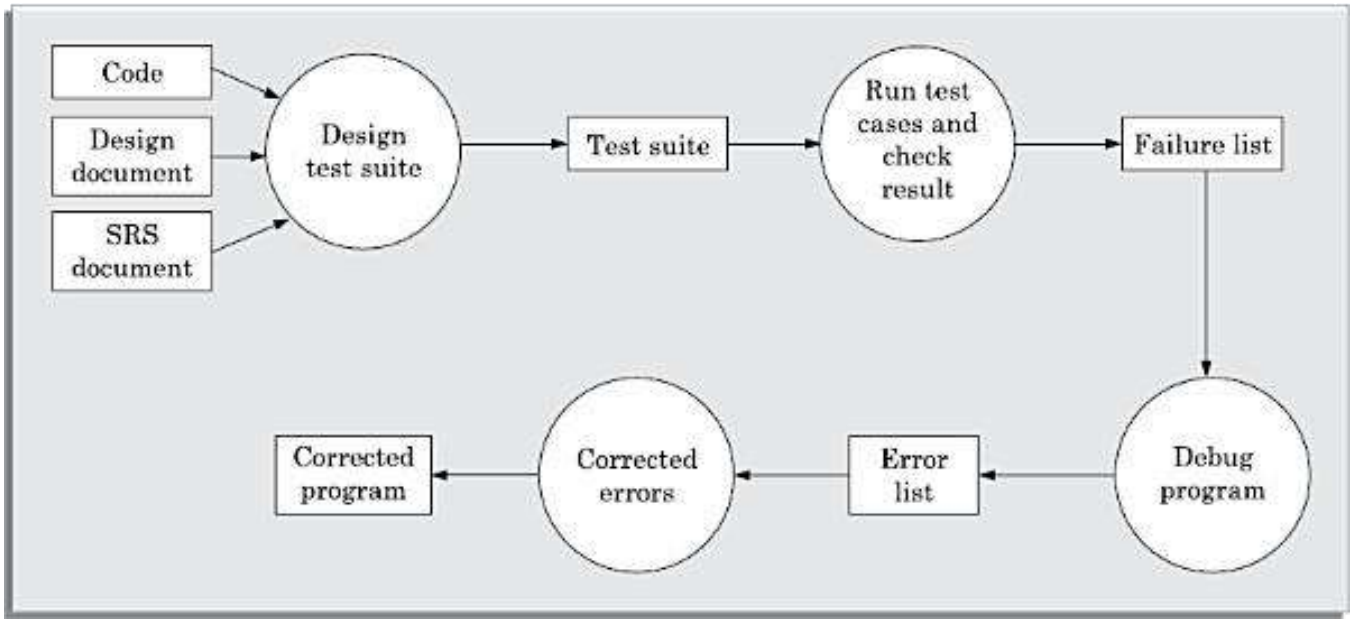


Figure 10.2: Testing process.

10.4.3 Why Design Test Cases?

Before discussing the various test case design techniques, we need to convince ourselves on the following question. Would it not be sufficient to test a software using a large number of random input values? Why design test cases? The answer to this question—this would be very costly and at the same time very ineffective way of testing due to the following reasons:

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as functional testing. On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called structural testing. Black- box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

10.4.4 Testing in the Large versus Testing in the Small

A software product is normally tested in three levels or stages:

- Unit testing
- Integration testing
- System testing

During unit testing, the individual functions (or units) of a program are tested.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully

integrated system is tested (system testing). Integration and system testing are known as testing in the large.

Often beginners ask the question—"Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?" The answer to this question is the following—There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is always a good idea to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

10.5 BLACK-BOX TESTING

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases:

- Equivalence class partitioning
- Boundary value analysis

In the following subsections, we will elaborate these two test case design techniques.

10.5.1 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in

the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are [−∞,0], [11,+∞].

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is □-{A,B,C}, where □ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

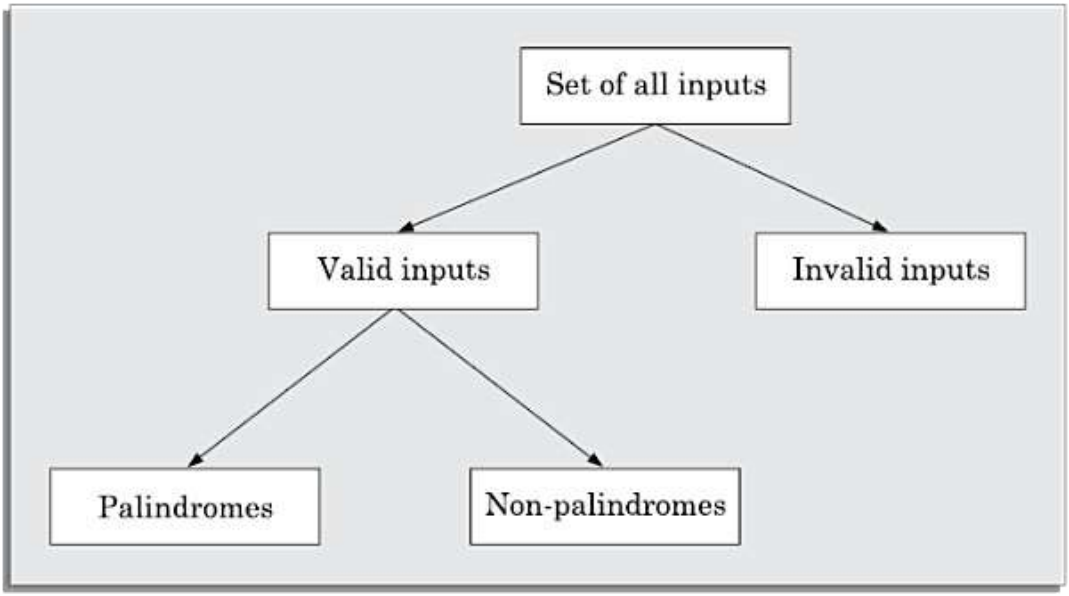


Figure 10.4: Equivalence classes for Example 10.6.

10.5.2 Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the

equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values(i.e., consist of a discrete collection of values) no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

10.5.3 Summary of the Black-box Test Suite Design Approach

We now summarise the important steps in the black-box test suite design approach:

- Examine the input and output values of the program.
- Identify the equivalence classes.
- Design equivalence class test cases by picking one representative value from each equivalence class.
- Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

The strategy for black-box testing is intuitive and simple. For black-box testing, the most important step is the identification of the equivalence classes. Often, the identification of the equivalence classes is not straightforward. However, with little practice one would be able to identify all equivalence classes in the input data domain. Without practice, one may overlook many equivalence classes in the input data set. Once the equivalence classes are identified, the equivalence class and boundary value test cases can be selected almost mechanically.

10.6 WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

10.6.1 Basic Concepts

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

> The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.

> A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

If a stronger testing has been performed, then a weaker testing need not be carried out.

10.6.2 Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.

10.6.3 Branch Coverage

A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed. Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

10.6.4 Multiple Condition Coverage

In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

10.6.5 Path Coverage

A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

A control flow graph describes how the control flows through the program. We can define a control flow graph as the following:

> A control flow graph describes the sequence in which the different instructions of a program get executed.

In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 10.5). There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other

node.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n □ N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.

10.6.6 McCabe's Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph ofa program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

How is path testing carried out by using computed McCabe's cyclomatic metric value?

Knowing the number of basis paths in a program does not make it any easier to design test cases for path coverage, only it gives an indication of the minimum number of test cases required for path coverage. For the CFG of a moderately complex program segment of say 20 nodes and 25 edges, you may need several days of effort to identify all the linearly independent paths in it and to design the test cases. It is therefore impractical to require the test designers to identify all the linearly independent paths in a code, and then design the test cases to force execution along each of the identified paths. In practice, for path testing, usually the tester keeps on forming test cases with random data and executes those until the required coverage is achieved. A testing tool such as a dynamic program analyser (see Section 10.8.2) is used to determine the percentage of linearly independent paths covered by the test cases that have been executed so far. If the percentage of linearly independent paths covered is below 90 per cent, more test cases (with random inputs) are added to increase the path coverage. Normally, it is not practical to target achievement of 100 per cent path coverage, since, the McCabe's metric is only an upper bound and does not give the exact number of paths.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).

3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. Repeat Test using a randomly designed set of test cases.
   Perform dynamic analysis to check the path coverage achieved.
   until at least 90 per cent path coverage is achieved.

10.6.7 Data Flow-based Testing

Data flow based testing method selects test paths of a program according to the definitions and uses of different variables in a program.

Consider a program P . For a statement numbered S of P , let

DEF(S) = {X /statement S contains a definition of X } and

USES(S)= {X /statement S contains a use of X }

For the statement S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. The definition of variable X at statement S is said to be live at statement S1 , if there exists a path from statement S to statement S1 which does not contain any definition of X .

All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition. All use criterion requires that all uses of a definition should be covered. Clearly, all-uses criterion is stronger than all-definitions criterion. An even stronger criterion is all definition-use-paths criterion, which requires the coverage of all possible definition-use paths that either are cycle-free or have only simple cycles. A simple cycle is a path in which only the end node and the start node are the same.

10.6.8 Mutation Testing

All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated

program and the change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors. A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement. A mutant may or may not cause an error in the program. If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.

10.7 DEBUGGING

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this Section, we shall summarise the important approaches that are available to identify the error locations. Each of these approaches has its own advantages anddisadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

10.7.1 Debugging Approaches

The following are some of the approaches that are popularly adopted by the programmers for debugging:

Brute force method

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly. Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

Backtracking

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002]. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

10.7.2 Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may

introduce new errors. Therefore after every round of error-fixing, regression testing (see Section 10.13) must be carried out.

10.8 INTEGRATION TESTING

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realise the full system. After each integration step, the partially integrated system is tested.

An important factor that guides the integration plan is the module dependency graph.

We have already discussed in Chapter 6 that a structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

- Big-bang approach to integration testing
- Top-down approach to integration testing
- Bottom-up approach to integration testing
- Mixed (also called sandwiched) approach to integration testing

In the following subsections, we provide an overview of these approaches to integration testing.

Big-bang approach to integration testing

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested. However, this technique can meaningfully be used only for very small systems. The main problem

with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix. As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

Top-down approach to integration testing

Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not

require any driver routines. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approach, testing can start as and when modules become available after unit testing. Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

10.8.1 Phased versus Incremental Integration Testing

Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Obviously, phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module. Please observe that a degenerate case of the phased integration testing approach is big-bang testing.

10.9 TESTING OBJECT-ORIENTED PROGRAMS

During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing. This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimised. However, it was soon realised that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs. The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that arepresent in procedural programs. Therefore additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following subsections.

10.9.1 What is a Suitable Unit for Testing Object-oriented Programs?

For procedural programs, we had seen that procedures are the basic units of testing. That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested. Thus, as far as procedural programs are concerned, procedures are the basic units of testing. Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing? Weyuker studied this issue and postulated his anticomposition axiom as follows:

> Adequate testing of individual methods does not ensure that a class has been satisfactorily tested.

The main intuitive justification for the anticomposition axiom is the following. A method operates in the scope of the data and other methods of its object. That is, all the methods share the data of the class. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant number of states. The behaviour of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily. A method has to be tested with all the other methods and data of the

corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.

An object is the basic unit of testing of object-oriented programs.

Thus, in an object oriented program, unit testing would mean testing each object in isolation. During integration testing (called cluster testing in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

10.9.2 Do Various Object-orientation Features Make Testing Easy?

In this section, we discuss the implications of different object-orientation features in testing.

Encapsulation: We had discussed in Chapter 7 that the encapsulation feature helps in data abstraction, error isolation, and error prevention. However, as far as testing is concerned, encapsulation is not an obstacle to testing, but leads to difficulty during debugging. Encapsulation prevents the tester from accessing the data internal to an object. Of course, it is possible that one can require classes to support state reporting methods to print out all the data internal to an object. Thus, the encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.

Inheritance: The inheritance feature helps in code reuse and was expected to simplify testing. It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features. However, this is not the case.

The reason for this is that the inherited methods would work in a new context (new data and method definitions). As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.

Dynamic binding: Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.

Object states: In contrast to the procedures in a procedural program,

objects store data permanently. As a result, objects do have significant states. The behaviour of an object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states. For example, when a book has been issued out in a library information system, the book reaches the issuedOut state. In this state, if the issue method is invoked, then it may not exhibit its normal behaviour.

In view of the discussions above, testing an object in only one of its states is not enough. The object has to be tested at all its possible states. Also, whether all the transitions between states (as specified in the object model) function properly or not should be tested. Additionally, it needs to be tested that no extra (sneak) transitions exist, neither are there extra states present other than those defined in the state model. For state-based testing, it is therefore beneficial to have the state model of the objects, so that the conformance of the object to its state model can be tested.

10.9.3 Why are Traditional Techniques Considered Not Satisfactory for Testing Object-oriented Programs?

We have already seen that in traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs. For example, statement coverage-based testing which is popular for testing procedural programs is not meaningful for object-oriented programs. The reason is that inherited methods have to be retested in the derived class. In fact, the different object- oriented features (inheritance, polymorphism, dynamic binding, state-based behaviour, etc.) require special test cases to be designed compared to the traditional testing as discussed in Section 10.11.4. The various object-orientation features are explicit in the design models, and it is usually difficult to extract from and analysis of the source code. As a result, the design model is a valuable artifact for testing object-oriented programs. Test cases are designed based on the design model. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a grey-box approach. Please note that grey-box testing is considered important for object-oriented programs. This is in contrast to testing procedural programs.

10.9.4 Grey-Box Testing of Object-oriented Programs

As we have already mentioned, model-based testing is important for object-oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.The following are some important types of grey-box testing that can be carried on based on UML models:

State-model-based testing

State coverage: Each method of an object are tested at each state of the object.

State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily.

State transition path coverage: All transition paths in the state model are tested.

Use case-based testing

Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.

Class diagram-based testing

Testing derived classes: All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derivec. lass, the inherited methods must be retested.

Association testing: All association relations are tested.

Aggregation testing: Various aggregate objects are created and tested.

Sequence diagram-based testing

Method coverage: All methods depicted in the sequence diagrams are covered. **Message path coverage:** All message paths that can be constructed from the sequence diagrams are covered.

10.9.5 Integration Testing of Object-oriented Programs

There are two main approaches to integration testing of object-oriented programs:

- Thread-based
- Use based

Thread-based approach: In this approach, all classes that need to collaborate to realise the behaviour of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested, another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.

Use-based approach: Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

10.9.6 Smoke Testing

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail. The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing. For smoke testing, a few test cases are designed to check whether the basic functionalities are working. For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

10.10 SOME GENERAL ISSUES ASSOCIATED WITH TESTING

In this section, we shall discuss two general issues associated with testing. These are—how to document the results of testing and how to perform regression testing.

Test documentation

A piece of documentation that is produced towards the end of testing is the test summary report. This report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem and their outcome. It normally specifies the following:

- What is the total number of tests that were applied to a subsystem.
- Out of the total number of tests how many tests were successful.
- How many were unsuccessful, and the degree to which they were unsuccessful, e.g., whether a test was an outright failure or whether

USER INTERFACE DESIGN

9.1 CHARACTERISTICS OF A GOOD USER INTERFACE

Before we start discussing anything about how to develop user interfaces, it is important to identify the different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one. In the following subsections, we identify a few important characteristics of a good user interface:

Speed of learning: A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorise commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:

— **U s e of metaphors**[1] **and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called metaphors. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it. Another popular metaphor is a shopping cart.

— **Consistency:** Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

— **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface discussed in Section 9.5.

The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

Speed of use: Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is some times referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface.

Speed of recall: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximised. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

Error prevention: A good user interface should minimise the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by an

average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users. Consistency of names, issue procedures, and behaviour of similar commands and the simplicity of the command issue procedures minimise error possibilities. Also, the interface should prevent the user from entering wrong values.

Aesthetic and attractive: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

Consistency: The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalise the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate

Feedback: A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his

request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.

Support for multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users.

Error recovery (undo facility): While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

9.2 TYPES OF USER INTERFACES

Broadly speaking, user interfaces can be classified into the following three categories:

- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

Each of these categories of interfaces has its own characteristic advantages and disadvantages. Therefore, most modern applications use a careful combination of all these three types of user interfaces for implementing the user command repertoire. It is very difficult to come up with a simple set of

guidelines as to which parts of the interface should be implemented using what type of interface. This choice is to a large extent dependent on the experience and discretion of the designer of the interface. However, a study of the basic characteristics and the relative advantages of different types of interfaces would give a fair idea to the designer regarding which commands should be supported using what type of interface. In the following three subsections, we briefly discuss some important characteristics, advantages, and disadvantages of using each type of user interface.

9.2.1 Command Language-based Interface

A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember. Thus, a command language-based interface can be made concise requiring minimal typing by the user. Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

Among the three categories of interfaces, the command language interface allows for most efficient command issue procedure requiring minimal typing. Further, a command language-based interface can be implemented even on cheap alphanumeric terminals. Also, a command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed. One can systematically develop a command language interface by using the standard compiler writing tools Lex and Yacc.

However, command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorise the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them. Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse. Obviously, for

casual and inexperienced users, command language-based interfaces are not suitable.

Issues in designing a command language-based interface

Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimise the total typing required. We elaborate these considerations in the following:

- The designer has to decide what mnemonics (command names) to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimise the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

9.2.2 Menu-based Interface

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Humans are much better in recognising something than recollecting it. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can

type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to design a menu-based interfae. A moderate-sized software might need hundreds or thousands of different menu choices. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms. In the following, we discuss some of the techniques available to structure a large number of menu items:

Scrolling menu: Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up or down to find the size he is looking for. However, if the commands do not have any definite ordering relation, then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organisation inefficient.

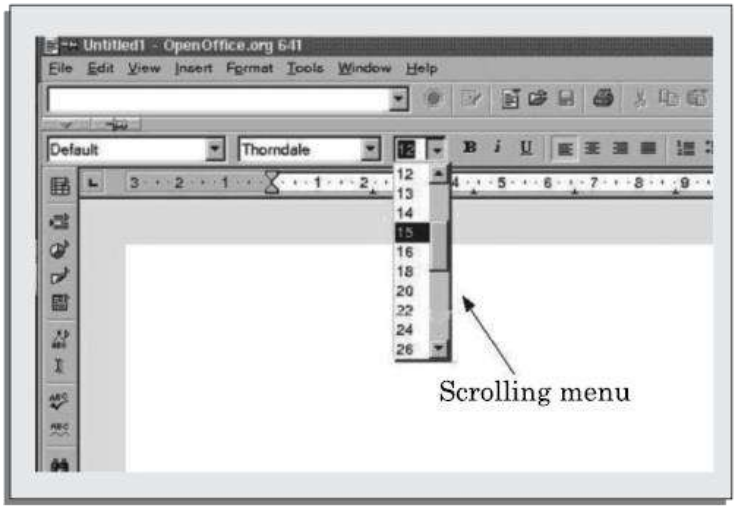


Figure 9.1: Font size selection using scrolling menu.

Walking menu: Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it

causes further menu items to be displayed adjacent to it in a sub-menu. An example of a walking menu is shown in Figure 9.2. A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after all limited.

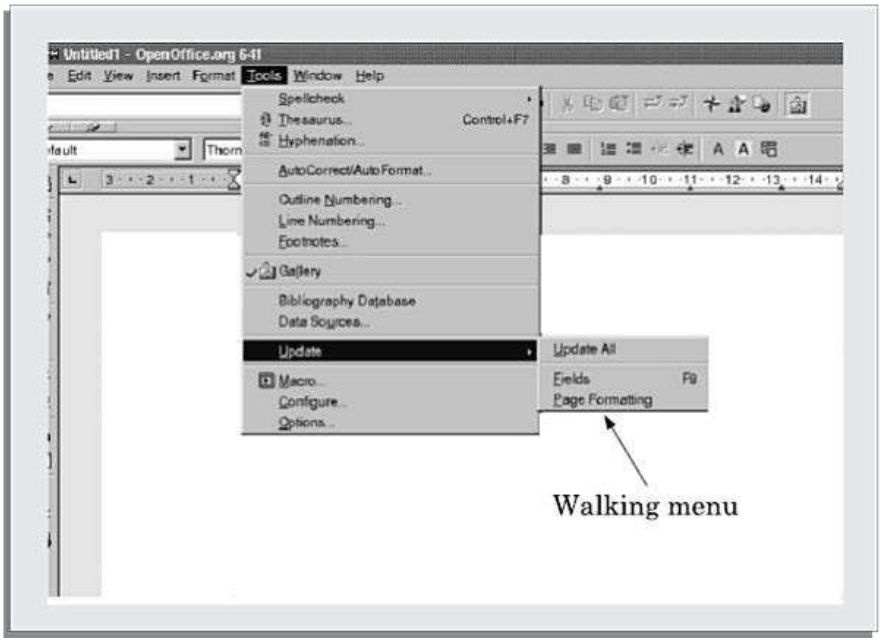


Figure 9.2: Example of walking menu.

Hierarchical menu: This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organised in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various sub-menu to form a hierarchical tree-like structure. Walking menu can be considered to be a form of hierarchical menu which is practicable when the tree is shallow. Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree. This probably is the main reason why this type of interface is very rarely used.

9.2.3 Direct Manipulation Interfaces

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons [2] or objects). For this reason, direct manipulation interfaces are sometimes called as iconic interfaces. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon

representing a file into an icon representing a trash box, for deleting the file.

Important advantages of iconic interfaces include the fact that the icons can be recognised by the users very easily, and that icons are language-independent. However, experienced users find direct manipulation interfaces very for too. Also, it is difficult to give complex commands using a direct manipulation interface. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files —which could be very easily done by issuing a command like delete *.*.

9.3 A USER INTERFACE DESIGN METHODOLOGY

At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface. What we present in this section is a set of recommendations which you can use to complement your ingenuity. Even though almost all popular GUI design methodologies are user-centered, this concept has to be clearly distinguished from a user interface design by users. Before we start discussing about the user interface design methodology, let us distinguish between a user-centered design and a design by users.

- User-centered design is the theme of almost all modern user interface design techniques. However, user-centered design does not mean design by users. One should not get the users to design the interface, nor should one assume that the user's opinion of which design alternative is superior is always right. Though users may have good knowledge of the tasks they have to perrform using a GUI, but they may not know the GUI design issues.
- Users have good knowledge of the tasks they have to perform, they also know whether they find an interface easy to learn and use but they have less understanding and experience in GUI design than the GUI developers.

9.3.1 Implications of Human Cognition Capabilities on User Interface Design

An area of human-computer interaction where extensive research has been conducted is how human cognitive capabilities and limitations influence the way an interface should be designed. In the following

subsections, we discuss some of the prominent issues that have been extensively reported in the literature.

Limited memory: Humans can remember at most seven unrelated items of information for short periods of time. Therefore, the GUI designer should not require the user to remember too many items of information at a time. It is the GUI designer's responsibility to anticipate what information the user will need at what point of each task and to ensure that the relevant information is displayed for the user to see. Showing the user some information at some point, and then asking him to recollect that information in a different screen where they no longer see the information, places a memory burden on the user and should be avoided wherever possible.

Frequent task closure: Doing a task (except for very trivial tasks) requires doing several subtasks. When the system gives a clear feedback to the user that a task has been successfully completed, the user gets a sense of achievement and relief. The user can clear out information regarding the completed task from memory. This is known as ta sk closure. When the overall task is fairly big and complex, it should be divided into subtasks, each of which has a clear subgoal which can be a closure point.

Recognition rather than recall. Information recall incurs a larger memory burden on the users and is to be avoided as far as possible. On the other hand, recognition of information from the alternatives shown to him is more acceptable.

Procedural versus ob ject-oriented: Procedural designs focus on tasks, prompting the user in each step of the task, giving them very few options for anything else. This approach is best applied in situations where the tasks are narrow and well-defined or where the users are inexperienced, such as a bank ATM. An object-oriented interface on the other hand focuses on objects. This allows the users a wide range of options.

9.3.2 A GUI Design Methodology

The GUI design methodology we present here is based on the seminal work of Frank Ludolph [Frank1998]. Our user interface design methodology consists of the following important steps:

- • Examine the use case model of the software. Interview, discuss, and review the GUI issues with the end-users.
- Task and object modelling.

- Metaphor selection.
- Interaction design and rough layout.
- Detailed presentation and graphics design.
- GUI construction.
- Usability evaluation.

Examining the use case model

We now elaborate the above steps in GUI design. The starting point for GUI design is the use case model. This captures the important tasks the users need to perform using the software. As far as possible, a user interface should be developed using one or more metaphors. Metaphors help in interface development at lower effort and reduced costs for training the users. Over time, people have developed efficient methods of dealing with some commonly occurring situations. These solutions are the themes of the metaphors. Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc. A solution based on metaphors is easily understood by the users, reducing learning time and training costs. Some commonly used metaphors are the following:

- White board
- Shopping cart
- Desktop
- Editor's work bench
- White page
- Yellow page
- Office cabinet
- Post box
- Bulletin board
- Visitor's Book

Task and ob ject modelling

A task is a human activity intended to achieve some goals. Examples of task goals can be as follows:

- Reserve an airline seat
- Buy an item
- Transfer money from one account to another

- Book a cargo for transmission to an address

A task model is an abstract model of the structure of a task. A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal. Each task can be modeled as a hierarchy of subtasks. A task model can be drawn using a graphical notation similar to the activity network model we discussed in Chapter 3. Tasks can be drawn as boxes with lines showing how a task is broken down into subtasks. An underlined task box would mean that no further decomposition of the task is required. An example of decomposition of a task into subtasks is shown inFigure 9.7.

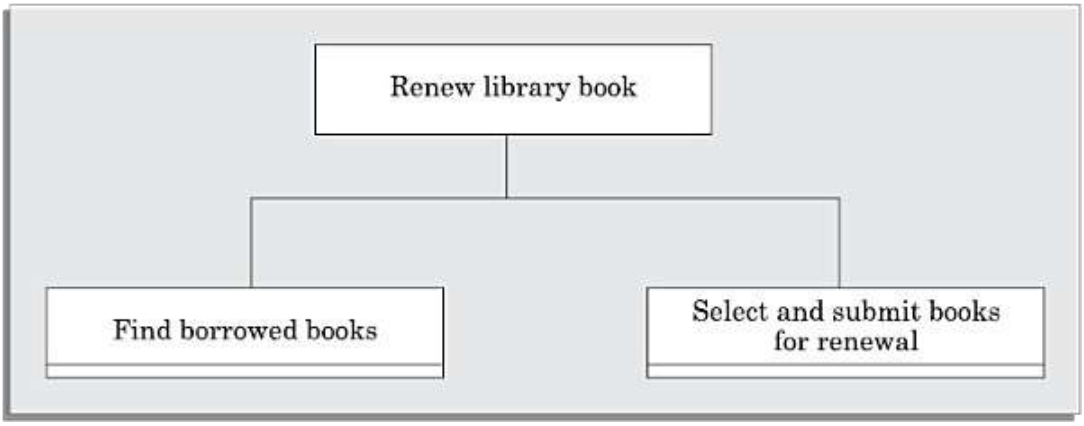


Figure 9.7: Decomposition of a task into subtasks.

Identification of the user objects forms the basis of an object-based design. A user object model is a model of business objects which the end-users believe that they are interacting with. The objects in a library software may be books, journals, members, etc. The objects in the supermarket automation software may be items, bills, indents, shopping list, etc. The state diagram for an object can be drawn using a notation similar to that used by UML (see Section 7.8). The state diagram of an object model can be used to determine which menu items should be dimmed in a state. An example state chart diagram for an order object is shown in Figure 9.8.

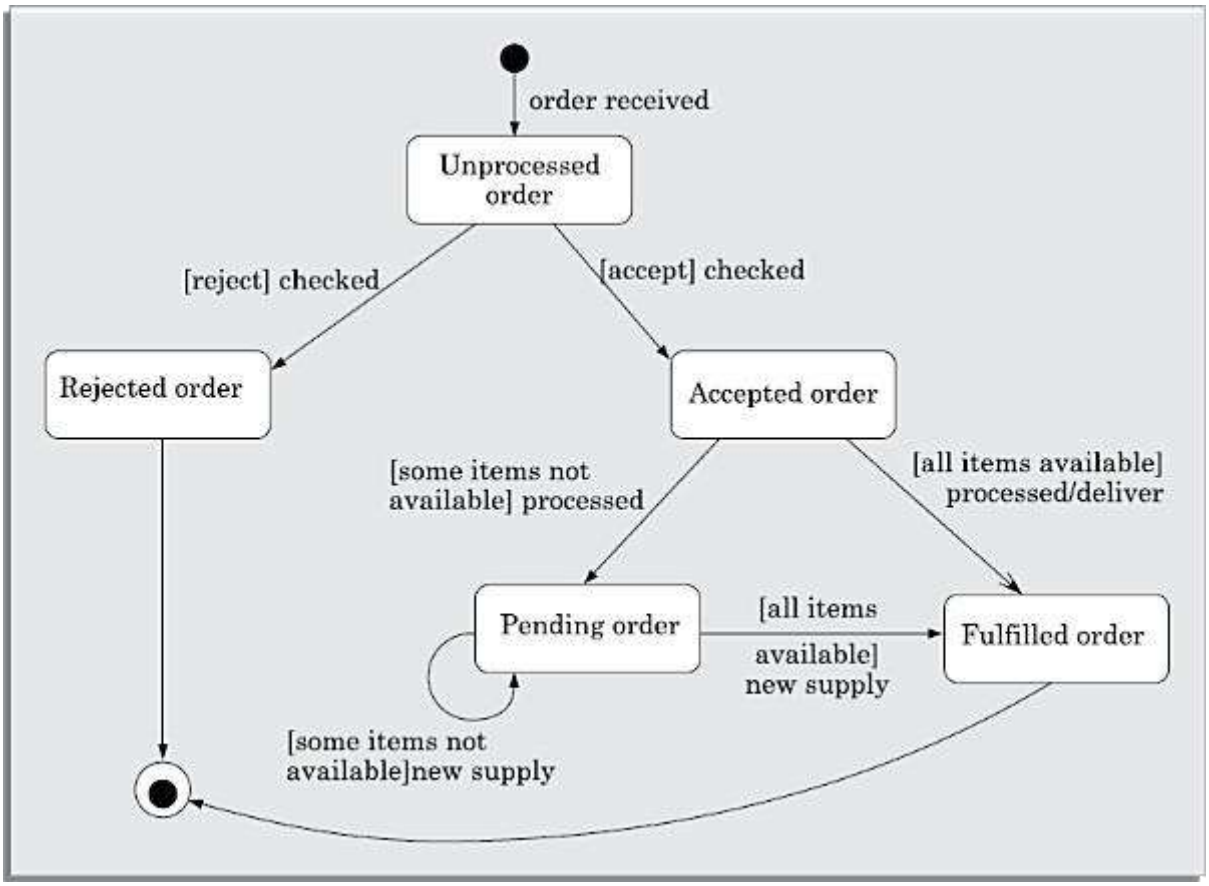


Figure 9.8: State chart diagram for an order object.

Metaphor selection

The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases. If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects. The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor. Another criterion that can be used to judge metaphors is that the metaphor should be as simple as possible, the operations using the metaphor should be clear and coherent and it should fit with the users' 'common sense' knowledge. For example, it would indeed be very awkward and a nuisance for the users if the scissor metaphor is used to glue different items.

Interaction design and rough layout

The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc. This involves making a choice from a set of available components that would best

suit the subtask. Rough layout concerns how the controls, an other widgets to be organised in windows.

Detailed presentation and graphics design

Each window should represent either an object or many objects that have a clear relationship to each other. At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing. At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window. This would force the user to move the cursor around the window to look for different objects.

GUI construction

Some of the windows have to be defined as modal dialogs. When a window is a modal dialog, no other windows in the application is accessible until the current window is closed. When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked. Modal dialogs are commonly used when an explicit confirmation or authorisation step is required for an action (e.g., confirmation of delete). Though use of modal dialogs are essential in some situations, overuse of modal dialogs reduces user flexibility. In particular, sequences of modal dialogs should be avoided.

User interface inspection

Nielson [Niel94] studied common usability problems and built a check list of points which can be easily checked for an interface. The following check list is based on the work of Nielson [Niel94]:

Visibility of the system status: The system should as far as possible keep the user informed about the status of the system and what is going on. For example, it should not be the case that a user gives a command and keeps waiting, wondering whether the system has crashed and he should reboot the system or that the results shall appear after some more time.

Match between the system and the real world: The system should speak the user's language with words, phrases, and concepts familiar to that used by the user, rather than using system-oriented terms.

Undoing mistakes: The user should feel that he is in control rather than feeling helpless or to be at the control of the system. An important step toward this is that the users should be able to undo and redo operations.

Consistency: The users should not have to wonder whether different words, concepts, and operations mean the same thing in different situations.

Recognition rather than recall: The user should not have to recall information which was presented in another screen. All data and instructions should be visible on the screen for selection by the user.

Support for multiple skill levels: Provision of accelerators for experienced users allows them to efficiently carry out the actions they most frequently require to perform.

Aesthetic and minimalist design: Dialogs and screens should not contain information which are irrelevant and are rarely needed. Every extra unit of information in a dialog or screen competes with the relevant units and diminishes their visibility.

Help and error messages: These should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggesting a solution.

Error prevention: Error possibilities should be minimised. A key principle in this regard is to prevent the user from entering wrong values. In situations where a choice has to be made from among a discrete set of values, the control should present only the valid values using a drop-down list, a set of option buttons or a similar multichoice control. When a specific format is required for attribute data, the entered data should be validated when the user attempts to submit the data.

FUNCTION-ORIENTED SOFTWARE DESIGN

6.1 OVERVIEW OF SA/SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

- During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

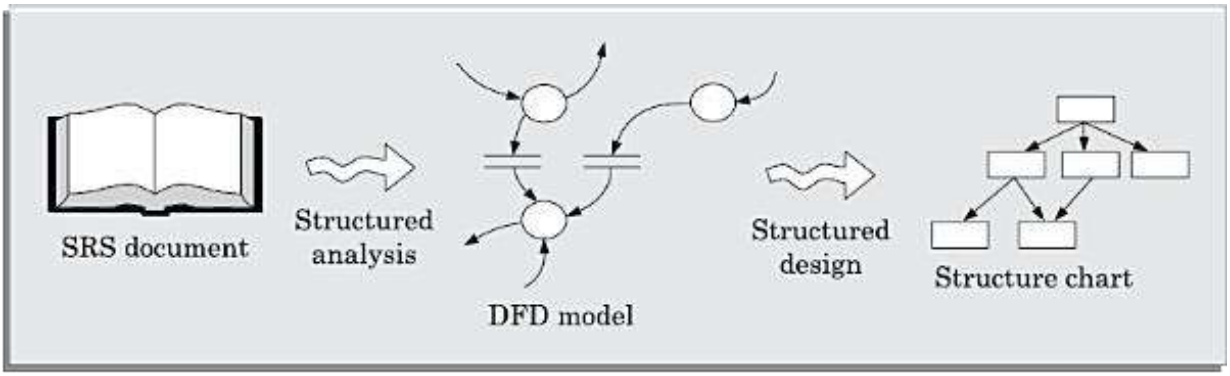


Figure 6.1: Structured analysis and structured design methodology.

As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.

The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

> It is important to understand that the purpose of structured analysis is to capture the detailed structure of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for implementation in some programming language.

The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user's terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.

In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

6.2 STRUCTURED ANALYSIS

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using data flow diagrams (DFDs).

DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.

Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used

by the functions, etc. In the DFD terminology, each function is called a process or a bubble. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.

DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organisation.

6.2.1 Data Flow Diagrams (DFDs)

The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system. The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism— it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams. We had pointed out while discussing the principle of abstraction in Section 1.3.2 that any hierarchical representation is an effective means to tackle complexity. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:

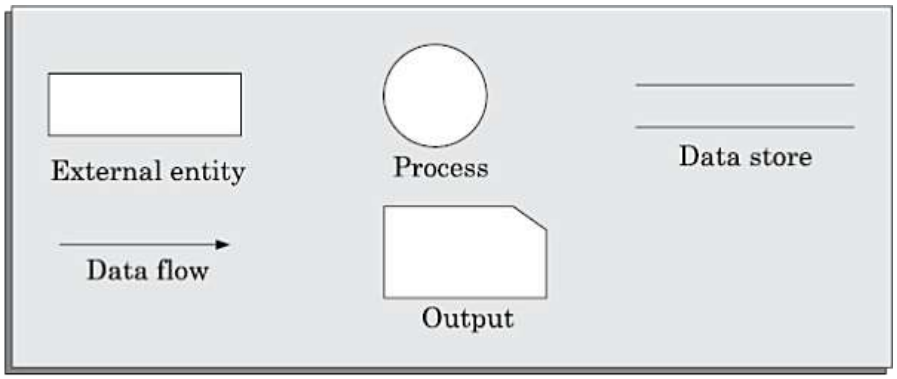


Figure 6.2: Symbols used for designing DFDs.

Function symbol: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).

External entity symbol: An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.

Data flow symbol: A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read-number to validate-number, data-item flowing into read-number, and valid-number flowing out of validate-number.

Data store symbol: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting t o a data store need not be annotated with the name of the corresponding data items. As an example of a data store, number is a data store in Figure 6.3(b).

Output symbol: The output symbol i s as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

The notations that we are following in this text are closer to the Yourdon's notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

Important concepts associated with constructing DFD models

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at t he same speed. An example of such an arrangement is shown in Figure 6.3(a). Here, the `validate-number` bubble can start processing only after t h e read-number bubble has supplied data to it; and the `read-number` bubble has to wait until the `validate-number` bubble has consumed its data.

However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

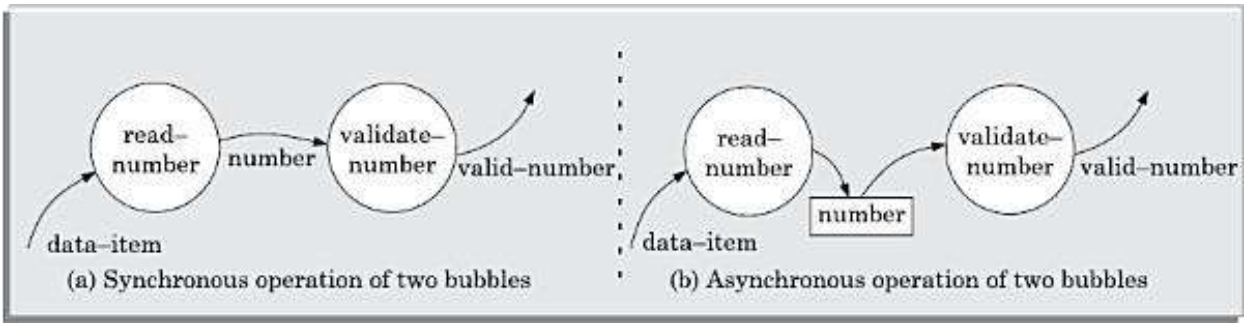


Figure 6.3: Synchronous and asynchronous data flow.

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary. A

data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc., as shown in Figure 6.4 discussed in new subsection. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

$$grossPay = regularPay + overtimePay$$

For the smallest units of data items, the data dictionary simply lists their name and their type. Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.

The dictionary plays a very important role in any software development process, especially for the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.

- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

For large systems, the data dictionary can become extremely complex and voluminous. Even moderate-sized projects can have thousands of entries in

the data dictionary. It becomes extremely di fficult to maintain a voluminous dictionary manually. Computer-aided software engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary. As a result, the designers do not have to spend almost any effort in creating the data dictionary. These CASE tools also support some query language facility to query about the definition and usage of data items. For example, queries may be formulated to determine which data item affects which processes, or a process affects which data items, or the definition and usage of specific data items, etc. Query handling is facilitated by storing the data dictionary in a relational database management system (RDBMS).

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

+: denotes composition of two data items, e.g. a+b represents data a and b.

[,,]: represents selection, i.e. any one of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs or b occurs.

(): the contents inside the bracket represent optional data which may or may not appear.

a+(b) represents either a or a+b occurs.

{}: represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

=: represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.

/* */: Anything appearing within /* and */ is considered as comment.

6.3 DEVELOPING THE DFD MODEL OF A SYSTEM

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.

The DFD model of a problem consists of many of DFDs and a single data dictionary.

The DFD model of a system i s constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system

(highest level). It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

6.3.1 Context Diagram

The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble. The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality. As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as 'Supermarket software'.

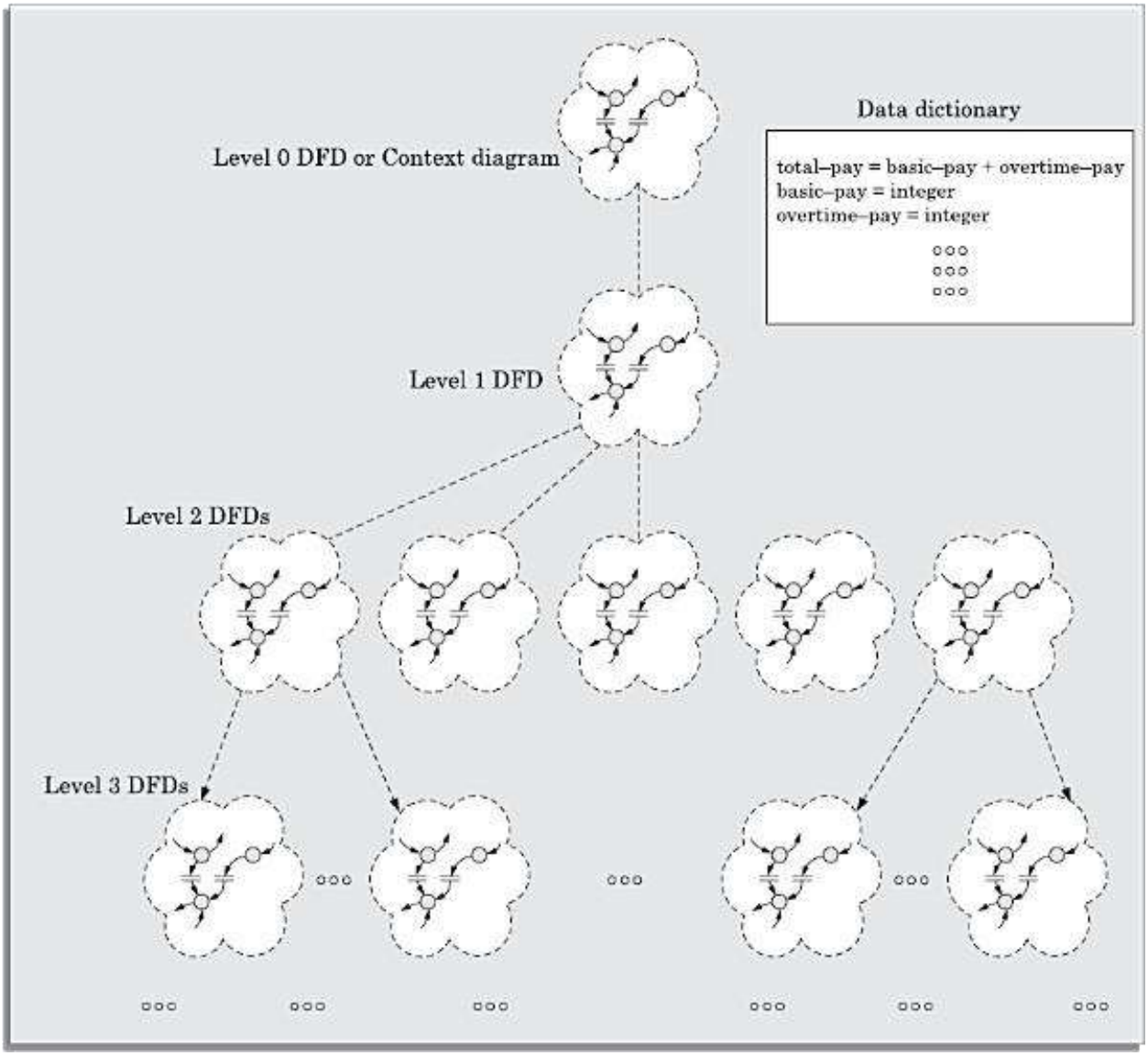


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

The context diagram establishes the context in which the system operates; that is, who are the users, what data do they input to the system, and what data they received by the system.

The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data

names.

To develop the context diagram of the system, we have to analyse the SRS document to identify the different types o f users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also includes any external systems which supply data to or receive data from the system.

6.3.2 Level 1 DFD

The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high-level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

What if a system has more than seven high-level requirements identified in the SRS document? In this case, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD. These can be split appropriately in the lower DFD levels. If a system has less than three high-level functional requirements, then some of the high-level requirements need to be split into their subfunctions so that we have roughly about five to seven bubbles represented on the diagram. We illustrate construction of level 1 DFDs in Examples 6.1 to 6.4.

Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring o r exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level m a k e that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles (i.e. more than seven bubbles) at any level o f a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried

on until a level is reached at which the function of the bubble can be described using a simple algorithm.

We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:
   - Different high-level functions that the system needs to perform.
   - Data input to every high-level function.
   - Data output from every high-level function.
   - Interactions (data flow) among the identified high-level functions.

   Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.

   Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

   Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:
   - Identify the different subfunctions of the high-level function.
   - Identify the data input to each of these subfunctions.
   - Identify the data output from each of these subfunctions.
   - Identify the interactions (data flow) among these subfunctions.

   Represent these aspects in a diagrammatic form using a DFD.

   Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is

decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing DFDs

The DFD model of a system usually consists of many DFDs that are organised in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD.

> The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.

We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1,0.1.2,0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d 2 flows in. Please note that dangling arrows (d1,d2,d3) represent the data flows into or out of a diagram.

How far to decompose?

A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions. For simple problems, decomposition up to level 1 should suffice. However, large industry standard problems may need decomposition up to level 3 or level 4. Rarely, if ever, decomposition beyond level 4 is needed.

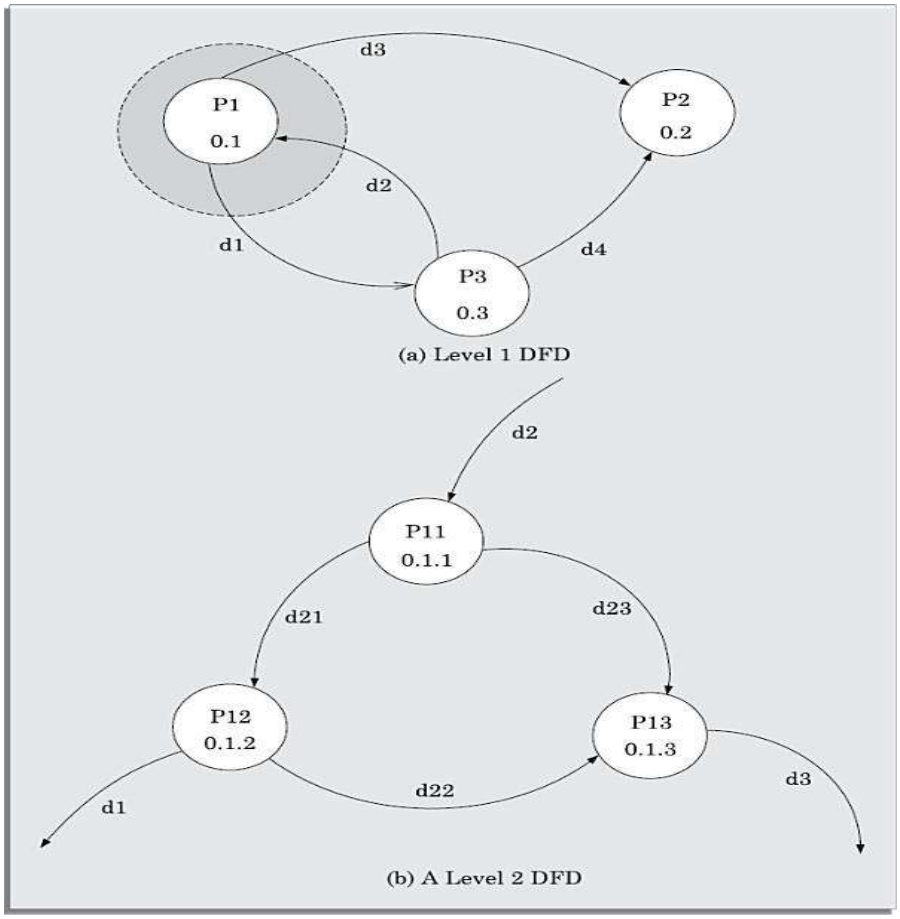


Figure 6.5: An example showing balanced decomposition.

Commonly made errors while constructing a DFD model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore useful to understand the different types of mistakes that beginners usually make while constructing the DFD model

of systems, so that you can consciously try to avoid them.The errors are as follows:

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.

- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

It is important to realise that a DFD represents only data flow, and it does not represent any control information.

The following are some illustrative mistakes of trying to represent control aspects such as:

Illustration 1. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While developing the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in Figure 6.6) to indicate that the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

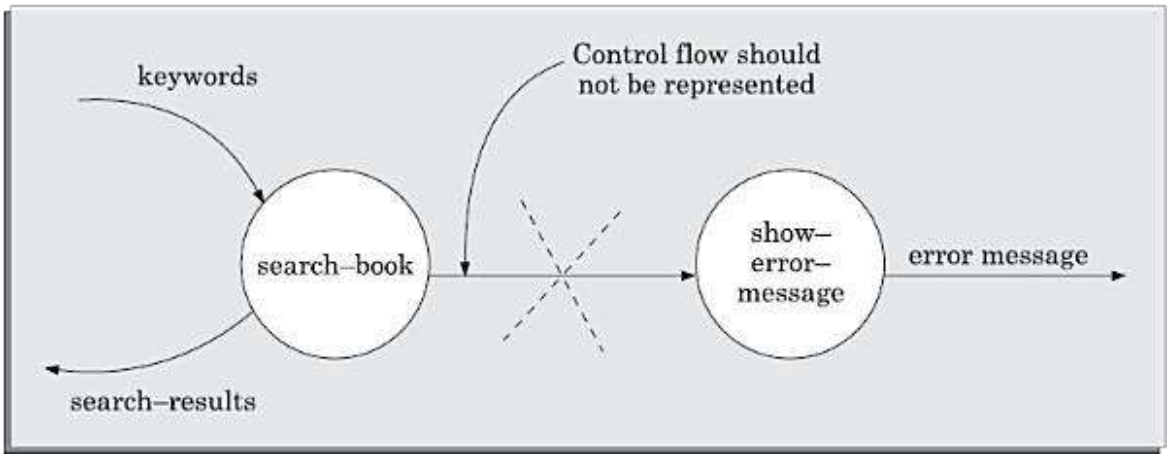


Figure 6.6: It is incorrect to show control information on a DFD.

Illustration 2. Another type of error occurs when one tries to represent when or in what order different functions (processes) are invoked. A DFD similarly should not represent the conditions under which different functions are invoked.

Illustration 3. If a bubble A invokes either the bubble B or the bubble C

depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data flow arrow should not connect two data stores or even a data store with an external entity. Thus, data cannot flow from a data store to another data store or to an external entity without any intervening processing. As a result, a data store should be connected only to bubbles through data flow arrows.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in the SRS document of the system should be overlooked.
- Only those functions of the system specified in the SRS document should be represented. That is, the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
- Incomplete data dictionary and data dictionary showing incorrect composition of data items are other frequently committed mistakes.
- The data and function names must be intuitive. Some students and even practicing developers use meaningless symbolic data names such as a,b,c, etc. Such names hinder understanding the DFD model.
- Novices usually clutter their DFDs with too many data flow arrow. It becomes difficult to understand a DFD if any bubble is associated with more than seven data flows. When there are too many data flowing in or out of a DFD, it is better to combine these data items into a high-level data item. Figure 6.7 shows an example concerning how a DFD can be simplified by combining several data flows into a single high-level data flow.

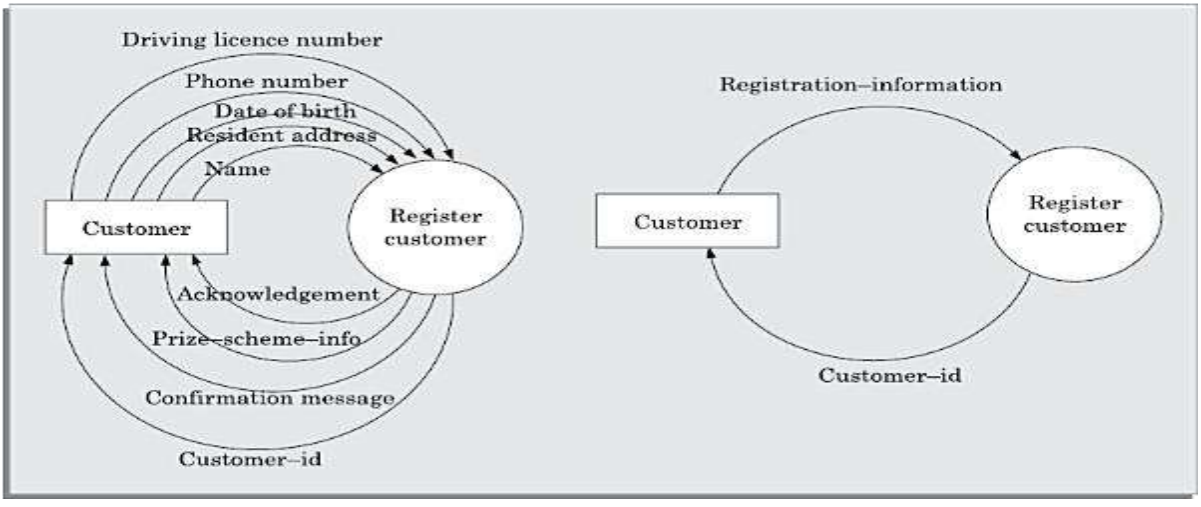


Figure 6.7: Illustration of how to avoid data cluttering.

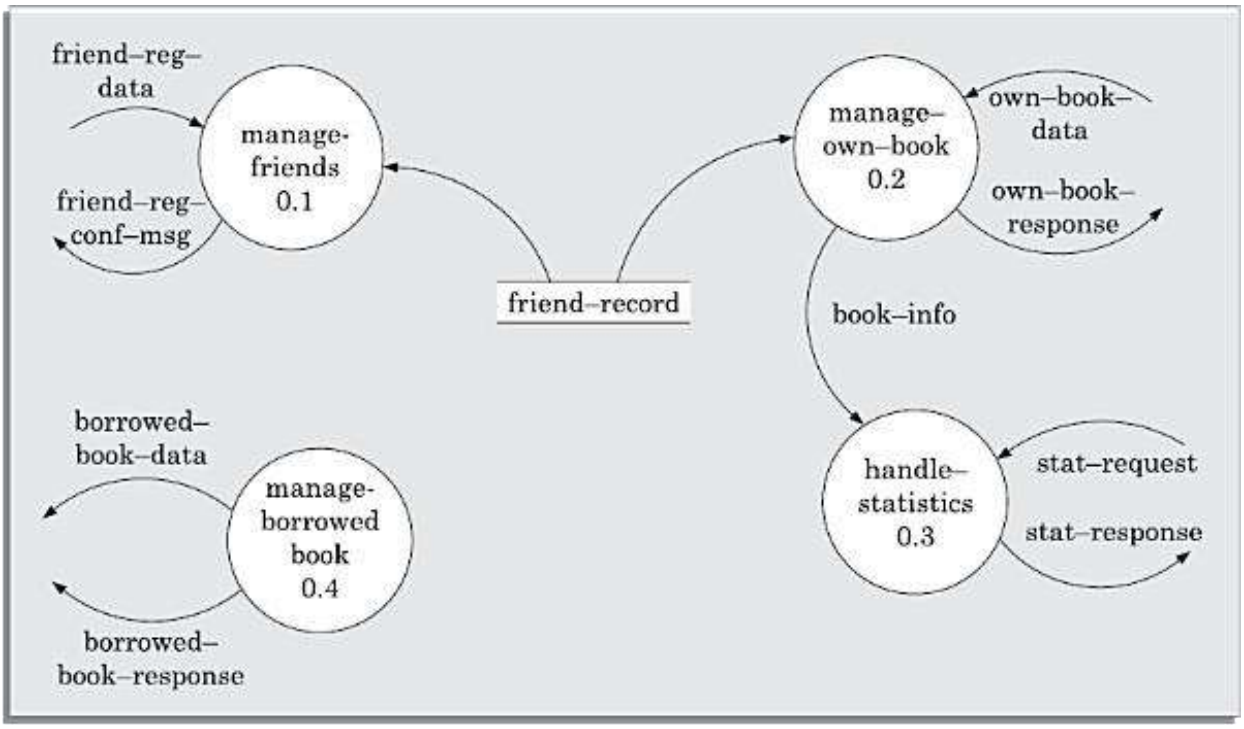


Figure 6.16: Level 1 DFD for Example 6.5.

The level 2 DFD for the manageOwnBook bubble is shown in Figure 6.17.

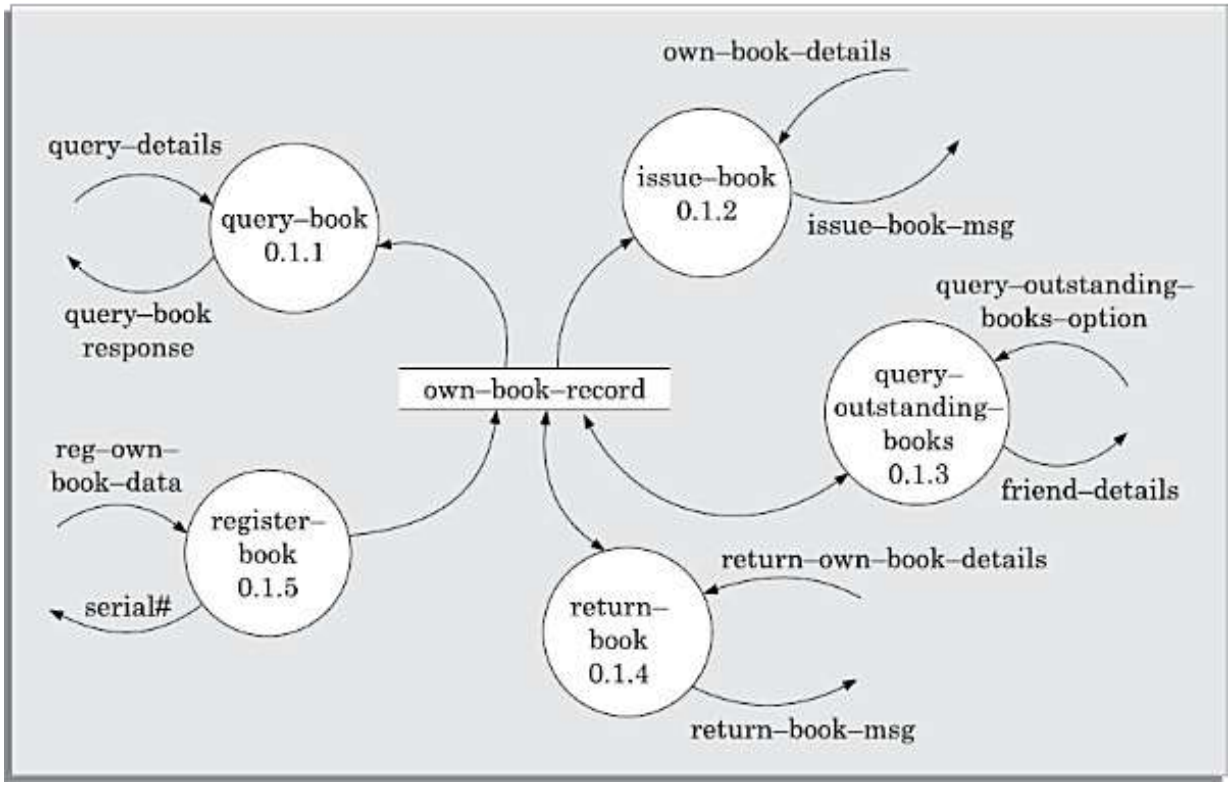


Figure 6.17: Level 2 DFD for Example 6.5.

6.3.3 Extending DFD Technique to Make it Applicable to Real-time Systems

In a real-time system, some of the high-level functions are associated with deadlines. Therefore, a function must not only produce correct results but also should produce them by some prespecified time. For real-time systems, execution time is an important consideration for arriving at a correct design. Therefore, explicit representation of control and event flow aspects are essential. One of the widely accepted techniques for extending the DFD technique to real-time system analysis is the Ward and Mellor technique [1985]. In the Ward and Mellor notation, a type of process that handles only control flows is introduced. These processes representing control processing are denoted using dashed bubbles. Control flows are shown using dashed lines/arrows.

Unlike Ward and Mellor, Hatley and Pirbhai [1987] show the dashed and solid representations on separate diagrams. To be able to separate the data processing and the control processing aspects, a control flow diagram (CFD) is defined. This reduces the complexity of the diagrams. In order to link the data processing and control processing diagrams, a notational reference (solid bar) to a control specification is used. The CSPEC describes the following:

- The effect of an external event or control signal.
- The processes that are invoked as a consequence of an event.

Control specifications represents the behavior of the system in two different ways:

- It contains a state transition diagram (STD). The STD is a sequential specification of behaviour.
- It contains a program activation table (PAT). The PAT is a combinatorial specification of behaviour. PAT represents invocation sequence of bubbles in a DFD.

6.4 STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure

chart represents the software architecture. The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks using which structure charts are designed are as following:

Rectangular boxes: A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows: An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a modules calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flo w arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.

Library modules: A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.

Selection: The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.

Repetition: A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

In any structure chart, there should be one and only one module at the top, called the root. There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A. The main reason behind this

restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules. However, it is possible for t wo higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

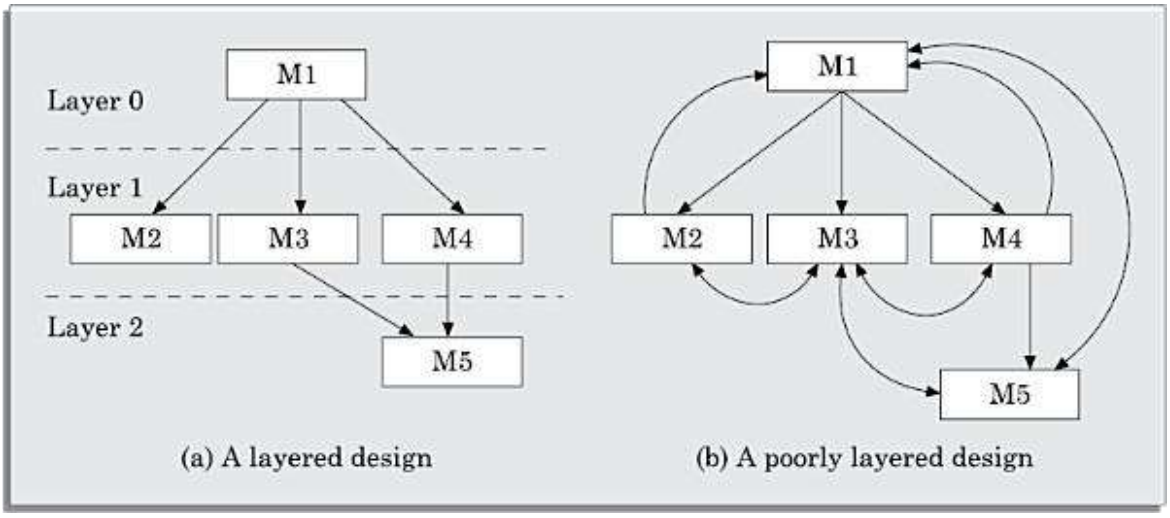


Figure 6.18: Examples of properly and poorly layered designs.

Flow chart ***versus*** structure chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flo w of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that i s inherent to a flow chart is suppressed in a structure chart.

6.4.1 Transformation of a DFD Model into Structure Chart

Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Transform analysis

Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:

- Input.
- Processing.
- Output.

The input portion in the DFD includes processes that transform input data from physical (e.g, character from terminal) to logical form (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the input and output parts requires experience and skill. One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms. Processes which sort input or filter data from it are central tansforms. T h e first level o f structure chart is produced by representing each input and output unit as a box and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialisation and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

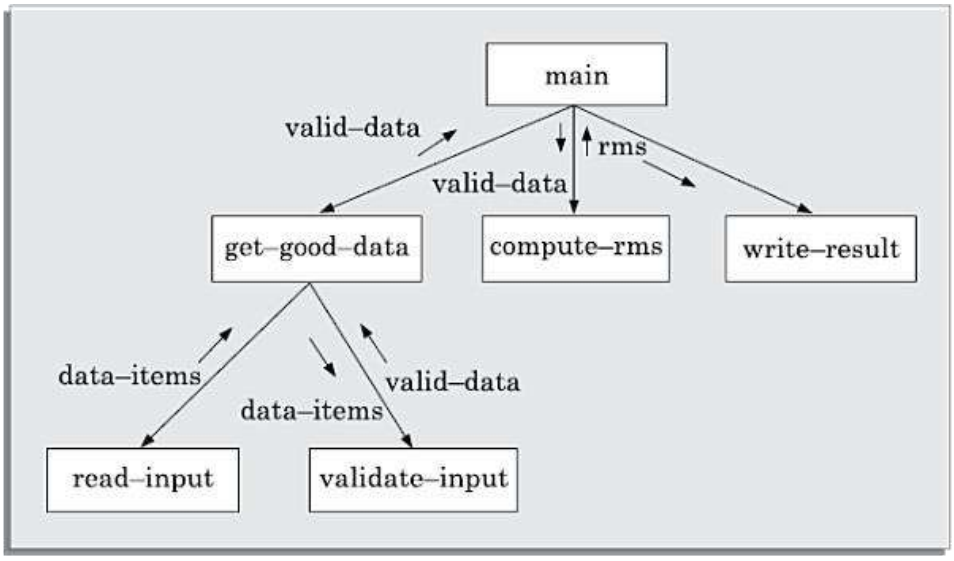


Figure 6.19: Structure chart for Example 6.6.

Transaction analysis

Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.

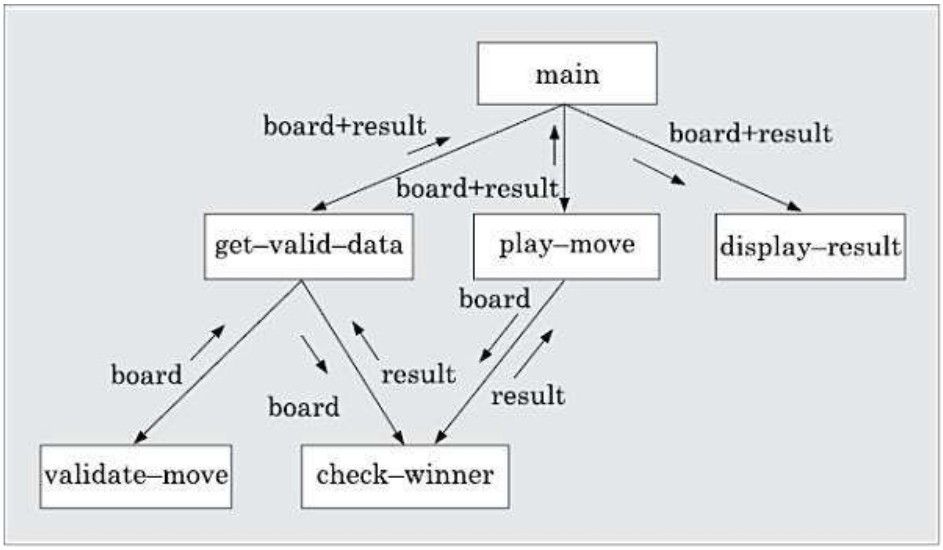


Figure 6.20: Structure chart for Example 6.7.

As in transform analysis, first all data entering into the DFD need to be identified. In a transaction-driven system, different data items may pass through different computation paths through the DFD. This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps. Each different way in which input data is processed is a transaction. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modules a nd a transaction-center module.

6.5 DETAILED DESIGN

During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules. The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

6.6 DESIGN REVIEW

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, members of the team

who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:

Traceability: Whether each bubble of the DFD can be traced to some module in the structure chart a nd vice versa. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and vice versa.

Correctness: Whether all the algorithms and data structures of the detailed design are correct.

Maintainability: Whether the design can be easily maintained in future.

Implementation: Whether the design can be easily and efficiently be implemented.

After the points raised by the reviewers is addressed by the designers, the design document becomes ready for implementation.

CODING AND TESTING

10.1 CODING

The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design. The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified. During the coding phase, different modules identified in the design document are coded according to their respective module specifications. We can describe the overall objective of the coding phase to be the following.

> The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.

Normally, good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard. These software development organisations formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously because of the significant business advantages it offers. The main advantages of adhering to a standard style of coding are the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It facilitates code understanding and code reuse.
- It promotes good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, the error return conventions, etc. Besides the coding standards, several coding guidelines are also prescribed by software companies. But, what is the difference between a coding guideline and a coding standard?

After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. It is important to detect as many errors as possible

during code reviews, because reviews are an efficient way of removing errors from code as compared to defect elimination using testing. We first discuss a few representative coding standards and guidelines.

10.2 CODE REVIEW

Testing is an effective defect removal mechanism. However, testing is applicable to only executable code. Review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing. Over the years, review techniques have become extremely popular and have been generalised for use with other work products.

Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Obviously, code review does not target to design syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors. Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.

The reason behind why code review is a much more cost-effective strategy to eliminate errors from code compared to testing is that reviews directly detect errors. On the other hand, testing only helps detect failures and significant effort is needed to locate the error during debugging.

The rationale behind the above statement is explained as follows. Eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time consuming activity. In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error.

Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss

these two code review techniques.

10.2.1 Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).

The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.

Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. These guidelines are based on personal experience, common sense, several other subjective factors. Therefore, these guidelines should be considered as examples rather than as accepted rules to be applied dogmatically. Some of these guidelines are following:

- The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
- In order to foster co-operation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

10.2.2 Code Inspection

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs. We can state the principal aim of the code inspection to be the following:

The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed to another programmer's errors.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more lik ely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.

Good software development companies collect statistics regarding different types of errors that are commonly committed by their engineers and identify the types of errors most frequently committed. Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialised variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatch between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values.
- Dangling reference caused when the referenced memory has not been allocated.

10.2.3 Clean Room Testing

Clean room testing was pioneered at IBM. This type of testing relies

heavily on walkthroughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. It is interesting to note that the term cleanroom was first coined at IBM by drawing analogy to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing. The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors. Also testing- based error detection is efficient for detecting certain errors that escape manual inspection.

10.3 SOFTWARE DOCUMENTATION

When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are helpful in the following ways:

- Good documents help enhance understandability of code. As a result, the availability of good documents help to reduce the effort and time required for maintenance.
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover[1] problem. Even when an engineer leaves the organisation, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
- Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.

Different types of software documents can broadly be classified into the following:

Internal documentation: These are provided in the source code itself.

External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.

We discuss these two types of documentation in the next section.

10.3.1 Internal Documentation

Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following:

- Comments embedded in the source code.
- Use of meaningful variable names.
- Module and function headers.
- Code indentation.
- Code structuring (i.e., code decomposed into modules and functions).
- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

Out of these different types of internal documentation, which one is the most valuable for understanding a piece of code?

The above assertion, of course, is in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without much thought. For example, the following style of code commenting is not much of a help in understanding the code.

```
a=10; /* a made 10 */
```

A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments. Good software development organisations usually ensure good internal documentation by appropriately formulating their coding standards

and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

10.3.2 External Documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc. A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.

An important feature that is requierd of any good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

Gunning's fog index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group

o f words that can be independently pronounced. For example, the word "sentence" has three syllables ("sen", "ten", and "ce"). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

10.4 TESTING

The aim of program testing is to help realize identify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Consider a function taking a floating point number as argument. If a tester takes 1sec to type in a value, then even a million testers would not be able to exhaustively test it after trying for a million number of years. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

In this section, we will discuss a few basic concepts in program testing on which our subsequent discussions on program testing would be based.

How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction. A highly simplified view of program testing is schematically shown in Figure 10.1. The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs. Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers. For examples, a software might fail for a test case only

when a network connection is enabled.

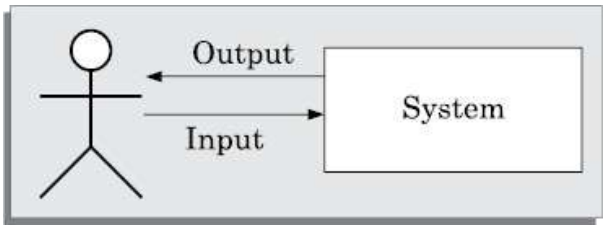


Figure 10.1: A simplified view of program testing.

Terminologies

As is true for any specialised domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.
- An **error** is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function.

Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community. Please note that in the domain of hardware testing, the term fault is used with a slightly different connotation [IEEE90] as compared to the terms error and bug.

Verification versus validation

The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different. We summarise the main differences between these two techniques in the following:

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- The primary techniques used for verification include review, simulation, formal verification, and testing. Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker. On the other hand, validation techniques are primarily based on product testing. Note that we have categorised testing both under program verification and validation. The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is a s per the module and module interface specifications. On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
- Verification does not require execution of the software, whereas

validation requires execution of the software.

- Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

10.4.2 Testing Activities

Testing involves performing the following main activities:

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques. We discuss a few important test case design techniques later in this Chapter.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analysed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected.Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

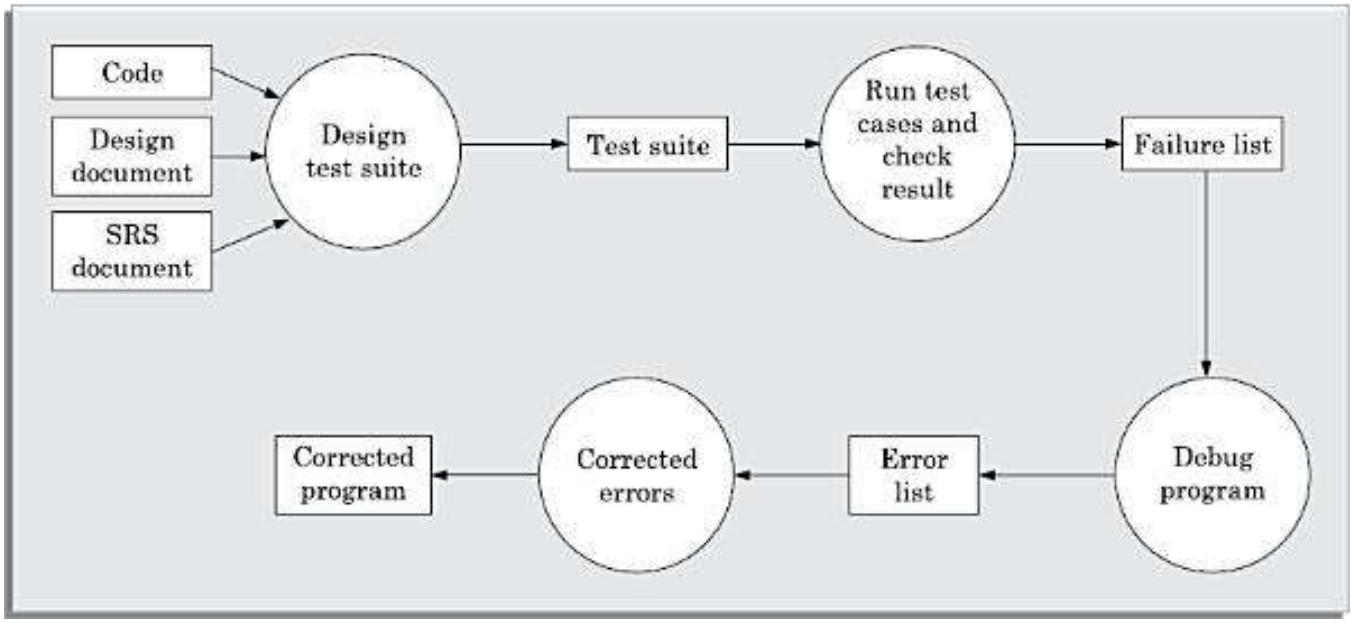


Figure 10.2: Testing process.

10.4.3 Why Design Test Cases?

Before discussing the various test case design techniques, we need to convince ourselves on the following question. Would it not be sufficient to test a software using a large number of random input values? Why design test cases? The answer to this question—this would be very costly and at the same time very ineffective way of testing due to the following reasons:

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as functional testing. On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called structural testing. Black- box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

10.4.4 Testing in the Large versus Testing in the Small

A software product is normally tested in three levels or stages:

- Unit testing
- Integration testing
- System testing

During unit testing, the individual functions (or units) of a program are tested.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully

integrated system is tested (system testing). Integration and system testing are known as testing in the large.

Often beginners ask the question—"Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?" The answer to this question is the following—There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is always a good idea to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

10.5 BLACK-BOX TESTING

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases:

- Equivalence class partitioning
- Boundary value analysis

In the following subsections, we will elaborate these two test case design techniques.

10.5.1 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in

the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are [−∞,0], [11,+∞].

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is □-{A,B,C}, where □ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

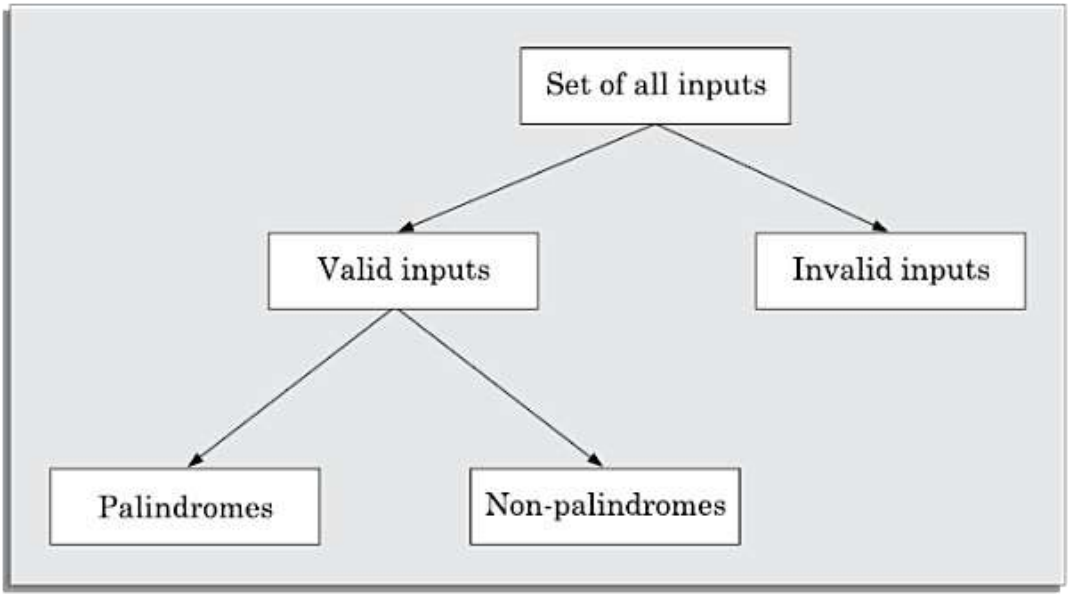


Figure 10.4: Equivalence classes for Example 10.6.

10.5.2 Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the

equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values(i.e., consist of a discrete collection of values) no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

10.5.3 Summary of the Black-box Test Suite Design Approach

We now summarise the important steps in the black-box test suite design approach:

- Examine the input and output values of the program.
- Identify the equivalence classes.
- Design equivalence class test cases by picking one representative value from each equivalence class.
- Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

The strategy for black-box testing is intuitive and simple. For black-box testing, the most important step is the identification of the equivalence classes. Often, the identification of the equivalence classes is not straightforward. However, with little practice one would be able to identify all equivalence classes in the input data domain. Without practice, one may overlook many equivalence classes in the input data set. Once the equivalence classes are identified, the equivalence class and boundary value test cases can be selected almost mechanically.

10.6 WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

10.6.1 Basic Concepts

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

> The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.

> A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

If a stronger testing has been performed, then a weaker testing need not be carried out.

10.6.2 Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.

10.6.3 Branch Coverage

A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed. Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

10.6.4 Multiple Condition Coverage

In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3). A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

10.6.5 Path Coverage

A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

A control flow graph describes how the control flows through the program. We can define a control flow graph as the following:

> A control flow graph describes the sequence in which the different instructions of a program get executed.

In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 10.5). There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other

node.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n □ N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.

10.6.6 McCabe's Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph ofa program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

How is path testing carried out by using computed McCabe's cyclomatic metric value?

Knowing the number of basis paths in a program does not make it any easier to design test cases for path coverage, only it gives an indication of the minimum number of test cases required for path coverage. For the CFG of a moderately complex program segment of say 20 nodes and 25 edges, you may need several days of effort to identify all the linearly independent paths in it and to design the test cases. It is therefore impractical to require the test designers to identify all the linearly independent paths in a code, and then design the test cases to force execution along each of the identified paths. In practice, for path testing, usually the tester keeps on forming test cases with random data and executes those until the required coverage is achieved. A testing tool such as a dynamic program analyser (see Section 10.8.2) is used to determine the percentage of linearly independent paths covered by the test cases that have been executed so far. If the percentage of linearly independent paths covered is below 90 per cent, more test cases (with random inputs) are added to increase the path coverage. Normally, it is not practical to target achievement of 100 per cent path coverage, since, the McCabe's metric is only an upper bound and does not give the exact number of paths.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).

3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. Repeat Test using a randomly designed set of test cases.
   Perform dynamic analysis to check the path coverage achieved.
   until at least 90 per cent path coverage is achieved.

10.6.7 Data Flow-based Testing

Data flow based testing method selects test paths of a program according to the definitions and uses of different variables in a program.

Consider a program P . For a statement numbered S of P , let

DEF(S) = {X /statement S contains a definition of X } and

USES(S)= {X /statement S contains a use of X }

For the statement S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. The definition of variable X at statement S is said to be live at statement S1 , if there exists a path from statement S to statement S1 which does not contain any definition of X .

All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition. All use criterion requires that all uses of a definition should be covered. Clearly, all-uses criterion is stronger than all-definitions criterion. An even stronger criterion is all definition-use-paths criterion, which requires the coverage of all possible definition-use paths that either are cycle-free or have only simple cycles. A simple cycle is a path in which only the end node and the start node are the same.

10.6.8 Mutation Testing

All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated

program and the change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors. A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement. A mutant may or may not cause an error in the program. If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.

10.7 DEBUGGING

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this Section, we shall summarise the important approaches that are available to identify the error locations. Each of these approaches has its own advantages anddisadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

10.7.1 Debugging Approaches

The following are some of the approaches that are popularly adopted by the programmers for debugging:

Brute force method

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly. Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

Backtracking

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. However, the search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002]. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

10.7.2 Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may

introduce new errors. Therefore after every round of error-fixing, regression testing (see Section 10.13) must be carried out.

10.8 INTEGRATION TESTING

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realise the full system. After each integration step, the partially integrated system is tested.

An important factor that guides the integration plan is the module dependency graph.

We have already discussed in Chapter 6 that a structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

- Big-bang approach to integration testing
- Top-down approach to integration testing
- Bottom-up approach to integration testing
- Mixed (also called sandwiched) approach to integration testing

In the following subsections, we provide an overview of these approaches to integration testing.

Big-bang approach to integration testing

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested. However, this technique can meaningfully be used only for very small systems. The main problem

with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix. As a result, big-bang integration testing is almost never used for large programs.

Bottom-up approach to integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

Top-down approach to integration testing

Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not

require any driver routines. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approach, testing can start as and when modules become available after unit testing. Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

10.8.1 Phased versus Incremental Integration Testing

Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Obviously, phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module. Please observe that a degenerate case of the phased integration testing approach is big-bang testing.

10.9 TESTING OBJECT-ORIENTED PROGRAMS

During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing. This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimised. However, it was soon realised that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs. The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that arepresent in procedural programs. Therefore additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following subsections.

10.9.1 What is a Suitable Unit for Testing Object-oriented Programs?

For procedural programs, we had seen that procedures are the basic units of testing. That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested. Thus, as far as procedural programs are concerned, procedures are the basic units of testing. Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing? Weyuker studied this issue and postulated his anticomposition axiom as follows:

> Adequate testing of individual methods does not ensure that a class has been satisfactorily tested.

The main intuitive justification for the anticomposition axiom is the following. A method operates in the scope of the data and other methods of its object. That is, all the methods share the data of the class. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant number of states. The behaviour of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily. A method has to be tested with all the other methods and data of the

corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.

An object is the basic unit of testing of object-oriented programs.

Thus, in an object oriented program, unit testing would mean testing each object in isolation. During integration testing (called cluster testing in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

10.9.2 Do Various Object-orientation Features Make Testing Easy?

In this section, we discuss the implications of different object-orientation features in testing.

Encapsulation: We had discussed in Chapter 7 that the encapsulation feature helps in data abstraction, error isolation, and error prevention. However, as far as testing is concerned, encapsulation is not an obstacle to testing, but leads to difficulty during debugging. Encapsulation prevents the tester from accessing the data internal to an object. Of course, it is possible that one can require classes to support state reporting methods to print out all the data internal to an object. Thus, the encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.

Inheritance: The inheritance feature helps in code reuse and was expected to simplify testing. It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features. However, this is not the case.

The reason for this is that the inherited methods would work in a new context (new data and method definitions). As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.

Dynamic binding: Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.

Object states: In contrast to the procedures in a procedural program,

objects store data permanently. As a result, objects do have significant states. The behaviour of an object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states. For example, when a book has been issued out in a library information system, the book reaches the issuedOut state. In this state, if the issue method is invoked, then it may not exhibit its normal behaviour.

In view of the discussions above, testing an object in only one of its states is not enough. The object has to be tested at all its possible states. Also, whether all the transitions between states (as specified in the object model) function properly or not should be tested. Additionally, it needs to be tested that no extra (sneak) transitions exist, neither are there extra states present other than those defined in the state model. For state-based testing, it is therefore beneficial to have the state model of the objects, so that the conformance of the object to its state model can be tested.

10.9.3 Why are Traditional Techniques Considered Not Satisfactory for Testing Object-oriented Programs?

We have already seen that in traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs. For example, statement coverage-based testing which is popular for testing procedural programs is not meaningful for object-oriented programs. The reason is that inherited methods have to be retested in the derived class. In fact, the different object- oriented features (inheritance, polymorphism, dynamic binding, state-based behaviour, etc.) require special test cases to be designed compared to the traditional testing as discussed in Section 10.11.4. The various object-orientation features are explicit in the design models, and it is usually difficult to extract from and analysis of the source code. As a result, the design model is a valuable artifact for testing object-oriented programs. Test cases are designed based on the design model. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a grey-box approach. Please note that grey-box testing is considered important for object-oriented programs. This is in contrast to testing procedural programs.

10.9.4 Grey-Box Testing of Object-oriented Programs

As we have already mentioned, model-based testing is important for object-oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.The following are some important types of grey-box testing that can be carried on based on UML models:

State-model-based testing

State coverage: Each method of an object are tested at each state of the object.

State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily.

State transition path coverage: All transition paths in the state model are tested.

Use case-based testing

Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.

Class diagram-based testing

Testing derived classes: All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derivec. lass, the inherited methods must be retested.

Association testing: All association relations are tested.

Aggregation testing: Various aggregate objects are created and tested.

Sequence diagram-based testing

Method coverage: All methods depicted in the sequence diagrams are covered. **Message path coverage:** All message paths that can be constructed from the sequence diagrams are covered.

10.9.5 Integration Testing of Object-oriented Programs

There are two main approaches to integration testing of object-oriented programs:

- Thread-based
- Use based

Thread-based approach: In this approach, all classes that need to collaborate to realise the behaviour of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested, another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.

Use-based approach: Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

10.9.6 Smoke Testing

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail. The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing. For smoke testing, a few test cases are designed to check whether the basic functionalities are working. For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

10.10 SOME GENERAL ISSUES ASSOCIATED WITH TESTING

In this section, we shall discuss two general issues associated with testing. These are—how to document the results of testing and how to perform regression testing.

Test documentation

A piece of documentation that is produced towards the end of testing is the test summary report. This report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem and their outcome. It normally specifies the following:

- What is the total number of tests that were applied to a subsystem.
- Out of the total number of tests how many tests were successful.
- How many were unsuccessful, and the degree to which they were unsuccessful, e.g., whether a test was an outright failure or whether

some of the expected results of the test were actually observed.

Regression testing

Regression testing spans unit, integration, and system testing. Instead, it is a separate dimension to these three forms of testing. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run — only those test cases that test the functions and are likely to be affected by the change need to be run. Whenever a software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.

SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

11.1 SOFTWARE RELIABILITY

The reliability of a software product essentially denotes its *trustworthiness or dependability*. Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced. It would have been very nice if we could mathematically characterise this relationship between reliability and the number of bugs present in the system using a simple closed form expression. Unfortunately, it is very difficult to characterise the observed reliability of a system in terms of the number of latent defects in the system using a simple mathematical expression. To get an insight into this issue, consider the following. Removing errors from those parts of a software product that are very infrequently executed, makes little difference to the perceived reliability of the product. It has been experimentally observed by analysing the behaviour of a large number of programs that 90 per cent of the execution time of a typical program is spent in executing only 10 per cent of the instructions in the program. The *most used* 10 per cent instructions are often called the *core*[1] of a program. The rest 90 per cent of the program statements are called *non-core* and are on the average executed only for 10 per cent of the total execution time. It therefore may not be very surprising to note that removing 60 per cent product defects from the least used parts of a system would typically result in only 3 per cent improvement to the product reliability. It is clear that the qua ntity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed. If an error is removed from an instruction that is frequently executed (i.e.,

belonging to the core of the program), then this would show up as a large improvement to the reliability figure. On the other hand, removing errors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.

Based on the above discussion we can say that reliability of a product depends not only on the number of latent errors but also on the the exact location of the errors. Apart from this, reliability also depends upon how the product is used, or on its *execution profile*. If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high.

On the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low. Different categories of users of a software product typically execute different functions of a software product.

Based on the above discussions, we can summarise the main reasons that make software reliability more difficult to measure than hardware reliability:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

In the following subsection, we shall discuss why software reliability measurement is a harder problem than hardware reliability measurement.

11.1.1 Hardware versus Software Reliability

An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.

A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug.

A comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product are sketched in Figure 11.1. Observe that the plot of change of reliability with time for a hardware component (Figure 11.1(a)) appears like a "bath tub". For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced.

The system then enters its useful life, where the rate of failure is almost constant. After some time (called product life time) the major components wear out, and the failure rate increases. The initial failures are usually covered through manufacturer's warranty.

In contrast to the hardware products, the software product show the highest failure rate just after purchase and installation (see the initial portion of the plot in Figure 11.1 (b)). As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

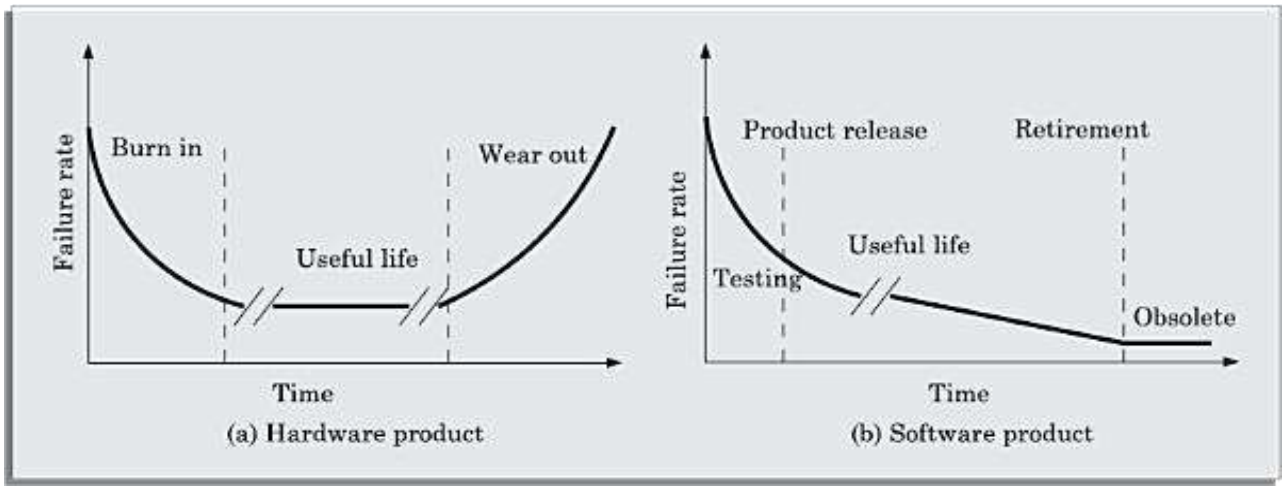


Figure 11.1: Change in failure rate of a product.

11.1.2 Reliability Metrics of Software Products

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the

software requirements specification (SRS) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very difficult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that correlate with reliability as follows:

Rate of occurrence of failure (ROCOF): ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.

Mean time to failure (MTTF): MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2, ..., t_n. Then, MTTF can be calculated as

$$\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}.$$

It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. are not taken into account in the time measurements and the clock is stopped at these times.

Mean time to repair (MTTR): Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean time between failure (MTBF): The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF

Probability of failure on demand (POFOD): Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made.

Availability: Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also

takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, and embedded controllers, etc.

11.1.3 Reliability Growth Modelling

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.

Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

Jelinski and Moranda model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in Figure 11.2. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that correction of different errors contribute differently to reliability growth.

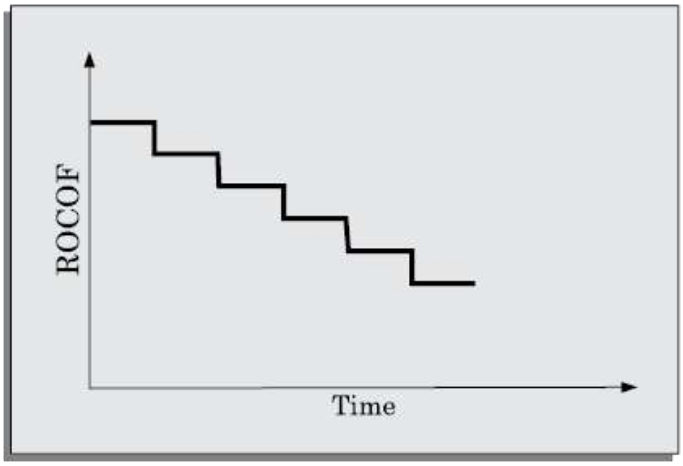


Figure 11.2: Step function model of reliability growth.

Littlewood and Verall's model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact

that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

There are more complex reliability growth models, which give more accurate approximations to the reliability growth. However, these models are out of scope of this text.

11.2 SOFTWARE QUALITY

Traditionally, the quality of a product is defined in terms of its fitness of purpose. That is, a good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document.

Although "fitness of purpose" is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.—"fitness of purpose" is not a wholly satisfactory definition of quality for software products.

Unlike hardware products, software lasts a long time, in the sense that it keeps evolving to accommodate changed circumstances. The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:

Portability: A software product is said to be portable, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.

Usability: A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

Reusability: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

Correctness: A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

11.3 SOFTWARE QUALITY MANAGEMENT SYSTEM

A quality management system (often referred to as quality system) is the principal methodology used by organisations to ensure that the products they develop have the desired quality. In the following subsections, we briefly discuss some of the important issues associated with a quality system:

Managerial structure and individual responsibilities

A quality system is the responsibility of the organisation as a whole. However, every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

Quality system activities

The quality system activities encompass the following:

- Auditing of projects to check if the processes are being followed.
- Collect process and product metrics and analyse them to check if quality goals are being met.
- Review of the quality system to make it more effective.
- Development of standards, procedures, and guidelines.
- Produce reports for the top management summarising the effectiveness of the quality system in the organisation.

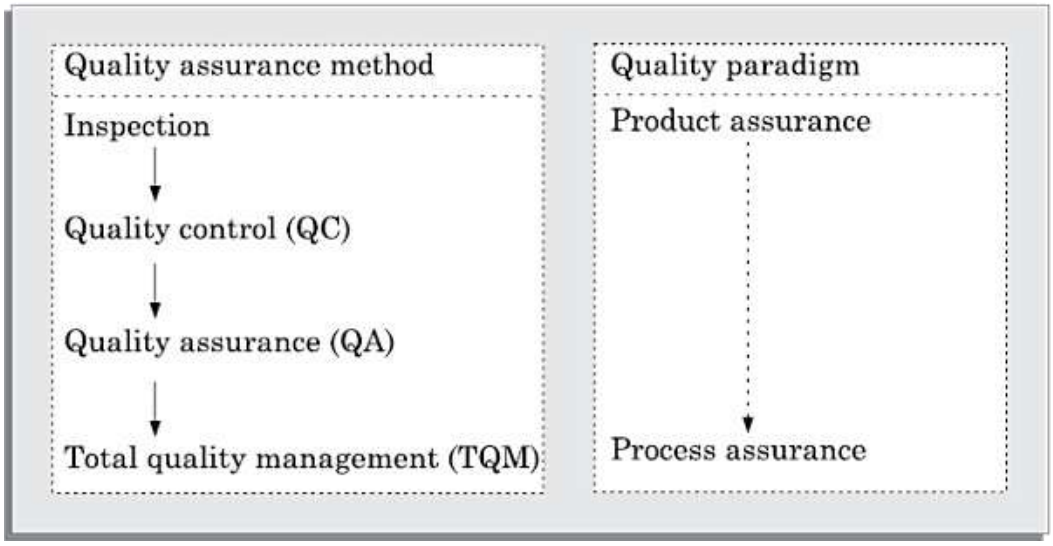
A good quality system must be well documented. Without a properly documented quality system, the application of quality controls and procedures become ad hoc, resulting in large variations in the quality of the products delivered. Also, an undocumented quality system sends clear messages to the staff about the attitude of the organisation towards quality assurance. International standards such as ISO 9000 provide guidance on how to organise a quality system.

11.3.1 Evolution of Quality Systems

Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality

systems of organisations have undergone four stages of evolution as shown in Figure 11.3. The initial product inspection method gave way to quality control (QC) principles.

Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.



Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products. The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.

The modern quality assurance paradigm includes guidance for recognising, defining, analysing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organisation must continuously be improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimising them through redesign.

A term related to TQM is business process re-engineering (BPR), which is aims at re-engineering the way business is carried out in an organisation, whereas our focus in this text is re-engineering of the software development process. From the above discussion, we can say that over the l a st six decades or so, the quality paradigm has shifted from product assurance to process assurance (see Figure 11.3).

11.3.2 Product Metrics versus Process Metrics

All modern quality systems lay emphasis on collection of certain product and process metrics during product development. Let us first understand the basic differences between product and process metrics.

Examples of product metrics are LOC and function point to measure size, PM (person- month) to measure the effort required to develop it, months to measure the time required to develop the product, time complexity of the algorithms, etc. Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

11.4 ISO 9000

International standards organisation (ISO) is a consortium of 63 countries established to formulate and foster standardisation. ISO published its 9000 series of standards in 1987.

11.4.1 What is ISO 9000 Certification?

ISO 9000 certification serves as a reference for contract between independent parties. In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not. In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system.

We have already seen that the quality system of an organisation applies to all its activities related to its products or services. The ISO standard addresses both operational aspects (that is, the process) and organisational aspects such as responsibilities, reporting, etc.

ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO 9003.

The types of software companies to which the different ISO standards apply are as follows:

ISO 9001: This standard applies to the organisations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.

ISO 9002: This standard applies to those organisations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organisations.

ISO 9003: This standard applies to organisations involved only in installation and testing of products.

11.4.2 ISO 9000 for Software Industry

ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company. Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organisations. An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of products. Two major differences between software development and development of other kinds of products are as follows:

- Software is intangible and therefore difficult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is difficult to control and manage anything that you cannot see and feel. In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to

accurately determine how much work has been completed and to estimate how much more time will it take.

- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. As an example, consider a steel making company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organisations.

11.4.3 Why Get ISO 9000 Certification?

There is a mad scramble among software development organisations for obtaining ISO certification due to the benefits it offers. Let us examine some of the benefits that accrue to organisations obtaining ISO certification:

- Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification. This is especially true in the international market. In fact, many organisations awarding international software development contracts insist that the development organisation have ISO 9000 certification. For this reason, it is vital for software organisations involved in software export to obtain ISO 9000 certification.
- ISO 9000 requires a well-documented software production process to be in place. A well- documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost-effective.

- ISO 9000 certification points out the weak points of an organisations and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

11.4.4 How to Get ISO 9000 Certification?

An organisation intending to obtain ISO 9000 certification applies to a ISO 9000 registrar for registration. The ISO 9000 registration process consists of the following stages:

Application stage: Once an organisation decides to go for ISO 9000 certification, it applies to a registrar for registration.

Pre-assessment: During this stage the registrar makes a rough assessment of the organisation.

Document review and adequacy audit: During this stage, the registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.

Compliance audit: During this stage, the registrar checks whether the suggestions made by it during review have been complied to by the organisation or not.

Registration: The registrar awards the ISO 9000 certificate after successful completion of all previous phases.

Continued surveillance: The registrar continues monitoring the organisation periodically.

This is probably due to the fact that the ISO 9000 certificate is issued for an organisation's process and not to any specific product of the organisation. An organisation using ISO certificate for product advertisements faces the risk of withdrawal of the certificate. In India, ISO 9000 certification is offered by BIS (Bureau of Indian Standards), STQC (Standardisation, testing, and quality control), and IRQS (Indian Register Quality System). IRQS has been accredited by the Dutch council of certifying bodies (RVC).

11.4.5 Summary of ISO 9001 Requirements

A summary of the main requirements of ISO 9001 as they relate of

software development are as follows:

Section numbers in brackets correspond to those in the standard itself:

Management responsibility (4.1)

- The management must have an effective quality policy.
- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality system (4.2)

A quality system must be maintained and documented.

Contract reviews (4.3)

Before entering into a contract, an organisation must review the contract to ensure that it is understood, and that the organisation has the necessary capability for carrying out its obligations.

Design control (4.4)

- The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
- Design inputs must be verified as adequate.
- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document control (4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)

Purchased material, including bought-in software must be checked for conforming to requirements.

Purchaser supplied product (4.7)

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product identification (4.8)

The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process control (4.9)

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and testing (4.10)

In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

Inspection, measuring and test equipment (4.11)

If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

Inspection and test status (4.12)

The status of an item must be identified. In software terms this implies configuration management and release control.

Control of non-conforming product (4.13)

In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

Corrective action (4.14)

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the

system needs improvement.

Handling (4.15)

This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality audits (4.17)

Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)

Training needs must be identified and met.

Various ISO 9001 requirements are largely common sense. Official guidance on the interpretation of ISO 9001 is inadequate at the present time, and taking expert advice is usually worthwhile.

11.4.6 Salient Features of ISO 9001 Requirements

In subsection 11.5.5 we pointed out the various requirements for the ISO 9001 certification. We can summarise the salient features all the the requirements as follows:

Document control: All documents concerned with the development of a software product should be properly managed, authorised, and controlled. This requires a configuration management system to be in place.

Planning: Proper plans should be prepared and then progress against these plans should be monitored.

Review: Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.

Testing: The product should be tested against specification.

Organisational aspects: Several organisational aspects should be addressed e.g., management reporting of the quality team.

11.4.7 ISO 9000-2000

ISO revised the quality standards in the year 2000 to fine tune the standards. The major changes include a mechanism for continuous process improvement. There is also an increased emphasis on the role of the top management, including establishing a measurable objectives for various roles and levels of the organisation. The new standard recognises that there can be many processes in an organisation.

11.5 SEI CAPABILITY MATURITY MODEL

SEI capability maturity model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book "Quality is Free"

In simple words, CMM is a reference model for apprising the software process maturity into different levels. This can be used to predict the most likely outcome to be expected from the next project that the organisation undertakes. It must be remembered that SEI CMM can be used in two ways—capability evaluation and software process assessment.

Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organisation. On the other hand, software process assessment is used by an organisation with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.

The different levels of SEI CMM have been designed so that it is easy for an organisation to slowly build its quality system starting from scratch. SEI CMM classifies software development industries into the following five maturity levels:

Level 1: Initial

A software development organisation at this level is characterised by ad hoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depend on individual efforts and heroics. When a developer leaves the organisation, the successor would have great difficulty in understanding

the process that was followed and the work completed. Also, no formal project management practices are followed. As a result, time pressure builds up towards the end of the delivery time, as a result short-cuts are tried out leading to low quality products.

Level 2: Repeatable

At this level, the basic project management practices such as tracking cost and schedule are established. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis, COCOMO, etc., are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Though there is a rough understanding among the developers about the process being followed, the process is not documented. Since the products are very similar, the success story on development of one product can repeated for another.

Level 3: Defined

At this level, the processes for both management and development activities are defined and documented. There is a common organisation-wide understanding of activities, roles, and responsibilities.The processes though defined, the process and product qualities are not measured. At this level, the organisation builds up the capabilities of its employees through periodic training programs. Also, review techniques are emphasized and documented to achieve phase containment of errors.

Level 4: Managed

At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimising

At this stage, process and product metrics are collected. Process and product measurement data are analysed for continuous process

improvement. At CMM level 5, an organisation would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organisation- wide. Level 5 organisations usually have a department whose sole responsibility is to assimilate latest tools and technologies and propagate them organisation-wide. Since the process changes continuously, it becomes necessary to effectively manage a changing process. Therefore, level 5 organisations use configuration management techniques to manage process changes.

The focus of each level and the corresponding key process areas are shown in the Table 11.1:

Table 11.1 Focus areas of CMM levels and Key Process Areas

CMM Level	*Focus*	*Key Process Areas (KPAs)*
Initial	Competent people	
Repeatable	Project management	Software project planning Software configuration management
Defined	Definition of processes	Process definition Training program Peer reviews
Managed	Product and process quality	Quantitative process metrics Software quality management
Optimising	Continuous process improvement	Defect prevention Process change management Technology change management

SEI CMM provides a list of key areas on which to focus to take an organisation from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one stage enhances the capability already built up.

However, the organisations trying out the CMM frequently face a problem that stems from the characteristic of the CMM itself.

CMM Shortcomings: CMM does suffer from several shortcomings. The important among these are the following:

- The most frequent complaint by organisations while trying out the CMM-based process improvement initiative is that they understand what is needed to be improved, but they need more guidance about how to improve it.
- Another shortcoming (that is common to ISO 9000) is that thicker documents, more detailed information, and longer meetings are

considered to be better. This is in contrast to the principles of software economics—reducing complexity and keeping the documentation to theminimum without sacrificing the relevant details.

- Getting an accurate measure of an organisation's current maturity level is also an issue. The CMM takes an activity-based approach to measuring maturity; if you do the prescribed set of activities then you are at a certain level. There is nothing that characterises or quantifies whether you do these activities well enough to deliver the intended results.

11.5.1 Comparison Between ISO 9000 Certification and SEI/CMM

Let us compare some of the key characteristics of ISO 9000 certification and the SEI CMM model for quality appraisal:

- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organisation in official documents, communication with external parties, and in tender quotations. However, SEI CMM assessment is purely for internal use.
- SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
- SEI CMM goes beyond quality assurance and prepares an organisation to ultimately achieve TQM. In fact, ISO 9001 aims at level 3 of SEI CMM model.
- SEI CMM model provides a list of key process areas (KPAs) on which an organisation at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement. In contrast, an organisation adopting ISO 9000 either qualifies for it or does not qualify.

11.5.2 Is SEI CMM Applicable to Small Organisations?

Highly systematic and measured approach to software development suits large organisations dealing with negotiated software, safety-critical software, etc. But, what about small organisations? These organisations typically handle applications such as small Internet, e-commerce applications, and often are without an established product range, revenue base, and experience on past projects, etc. For such organisations, a CMM-based appraisal is probably excessive. These

organisations need to operate more efficiently at the lower levels of maturity. For example, they need to practise effective project management, reviews, configuration management, etc.

11.5.3 Capability Maturity Model Integration (CMMI)

Capability maturity model integration (CMMI) is the successor of the capability maturity model (CMM). The CMM was developed from 1987 until 1997. CMMI aimed to improve the usability of maturity models by integrating many different models into one framework.

After CMMI was first released in 1990, it was adopted and used in many domains. For example, CMMs were developed for disciplines such as systems engineering (SE-CMM), people management (PCMM), software acquisition (SA-CMM), and others.

Although many organisations found these models to be useful, they also struggled with problems caused by overlap, inconsistencies, and integrating the models. In this context, CMMI is generalised to be applicable to many domains.

11.6 SIX SIGMA

General Electric (GE) corporation first began Six Sigma in 1995 after Motorola and Allied Signal blazed the Six Sigma trail. Since them, thousands of companies around the world have discovered the far reaching benefits of Six Sigma.

The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry.

Six Sigma at many organisations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and from product to service.

The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined

as any system behaviour that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator.

The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies—DMAIC and DMADV.

The Six Sigma DMAIC process (define, measure, analyse, improve, control) is an improvement system for existing processes falling below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyse, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

COMPUTER AIDED SOFTWARE ENGINEERING

12.1 CASE AND ITS SCOPE

We first need to define what is a CASE tool and what is a CASE environment. A CASE tool is a generic term used to denote any form of automated support for software engineering, In a more restrictive sense a CASE tool can mean any tool used to automate some activity associated with software development. Many CASE tools are now available. Some of these tools assist in phase-related tasks such asspecification, structured analysis, design, coding, testing, etc. and others to non-phase activities such as project management and configuration management. The primary objectives in using any CASE tool are:

- To increase productivity.
- To help produce better quality software at lower cost.

12.2 CASE ENVIRONMENT

Although individual CASE tools are useful, the true power of a tool set can be realised only when these set of tools are integrated into a common framework or environment. If the different CASE tools are not integrated, then the data generated by one tool would have to input to the other tools. This may also involve format conversions as the tools developed by different vendors are likely to use different formats. This results in additional effort of exporting data from one tool and importing to another. Also, many tools do not allow exporting data and maintain the data in proprietary formats.

CASE tools are characterised by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software. This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves.

Thus a CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development. The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger.

The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements arehighlighted. Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc. A schematic representation of a CASE environment is shown in Figure 12.1.

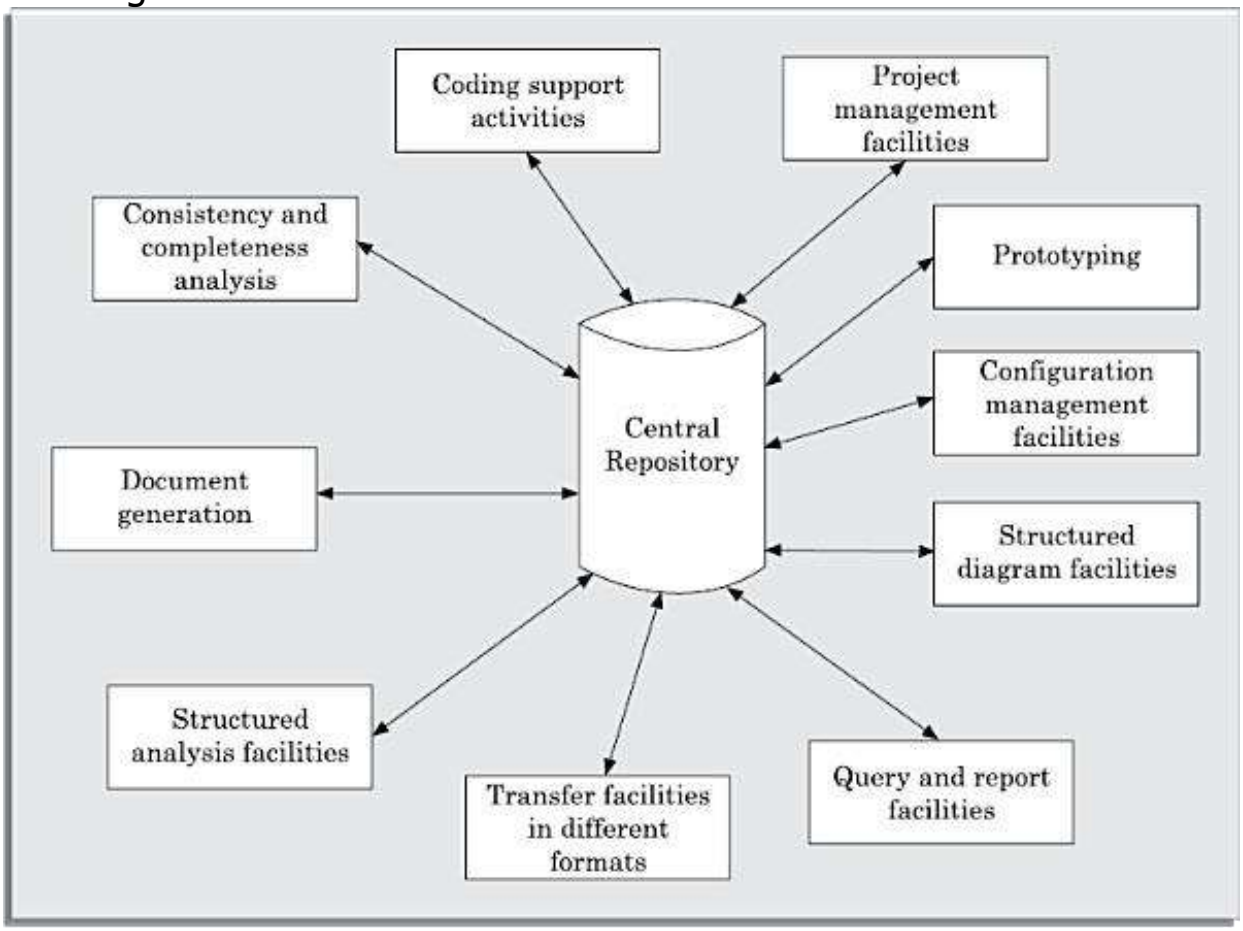


Figure 12.1: A CASE environment.

The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc., All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

12.2.1 Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases. Different studies carry out to measure the impact of CASE, put the effort reduction between 30 per cent and 40 per cent.
- Use of CASE tools leads to considerable improvements in quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development, and the chances of human error is considerably reduced.
- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced, and therefore, chances of inconsistent documentation is reduced to a great extent.
- CASE tools take out most of the drudgery in a software engineers work. For example, they need not check meticulously the balancing of the DFDs, but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

12.3 CASE SUPPORT IN SOFTWARE LIFE CYCLE

Let us examine the various types of support that CASE provides during the different phases of a software life cycle. CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases. Some of the possible support that CASE tools usually provide in the software development life cycle are discussed below.

12.3.1 Prototyping Support

We have already seen that prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The prototyping CASE tool's requirements are as follows:

- Define user interaction.
- Define the system control flow.
- Store and retrieve data required by the system.
- Incorporate some processing logic.

There are several stand alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary,help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.
- The run time system of prototype should support mock up run of the actual system and management of the input and output data.

12.3.2 Structured Analysis and Design

Several diagramming techniques are used for structured analysis and structured design. A CASE tool should support one or more of the structured analysis and design technique. The CASE tool should support effortlessly drawing analysis and design diagrams. The CASE tool should support drawing fairly complex diagrams and preferably through a hierarchy of levels. It should provide easy navigation through different

levels and through design and analysis. The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Wherever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there is heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

12.3.3 Code Generation

As far as code generation is concerned, the general expectation from a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic support expected from a CASE tool during code generation phase are the following:

- The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
- The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular programming languages.
- It should generate database tables for relational database management systems.
- The tool should generate code for user interface from prototype definition for X window and MS window based applications.

12.3.4 Test Case Generator

The CASE tool for test case generation should have the following features:

- It should support both design and requirement testing
- It should generate test set reports in ASCII format which can be directly imported into the test plan document.

12.4 ARCHITECTURE OF A CASE ENVIRONMENT

The architecture of a typical modern CASE environment is shown diagrammatically in Figure 12.2. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. We have already seen

the characteristics of the tool set. Let us examine the other components of a CASE environment.

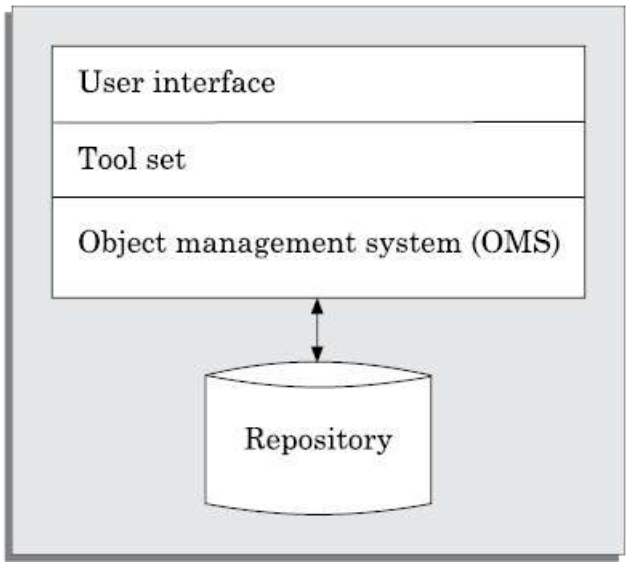


Figure 12.2: Architecture of a modern CASE environment.

User interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

Object management system and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping these entities into the underlying storage management system.

SOFTWARE MAINTENANCE

13.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts.

Types of Software Maintenance

There are three types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

13.1.1 Characteristics of Software Evolution

Lehman's first law: A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts. This law clearly shows that every product

irrespective of how well designed must undergo maintenance. In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law: The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that when you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support. If you do not redesign the system, the additions will be more complex that they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law: Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

13.1.2 Special Problems Associated with Software Maintenance

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:

Software maintenance work in organisations is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organisation often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work, and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of

software products needing maintenance are legacy products. Though the word legacy implies "aged" software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

13.2 SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

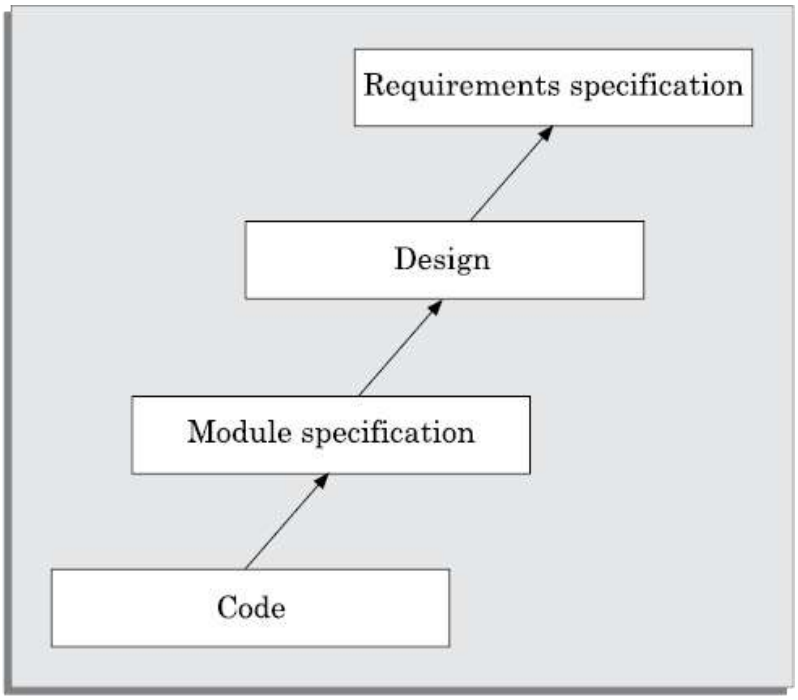


Figure 13.1: A process model for reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the proces of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

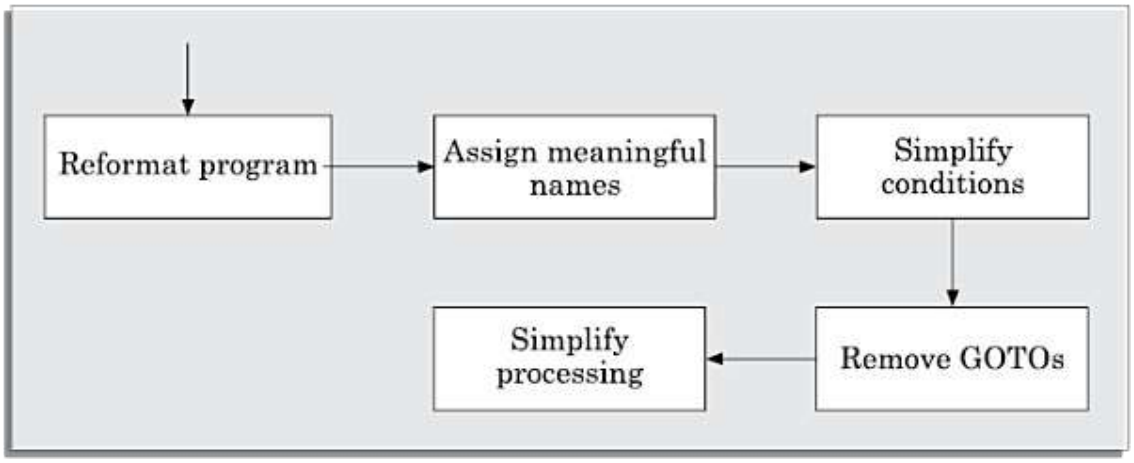


Figure 13.2: Cosmetic changes carried out before reverse engineering.

13.3 SOFTWARE MAINTENANCE PROCESS MODELS

Before discussing process models for software maintenance, we need to analyse various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as:

(i) the extent of modification to the product required,

(ii) the resources available to the maintenance team,

(iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.),

(iv) the expected project risks, etc. When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope (activities required) for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed.

First model

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Figure 13.3. In this approach, the project starts by gathering the requirements for changes. The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the

task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re-engineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.

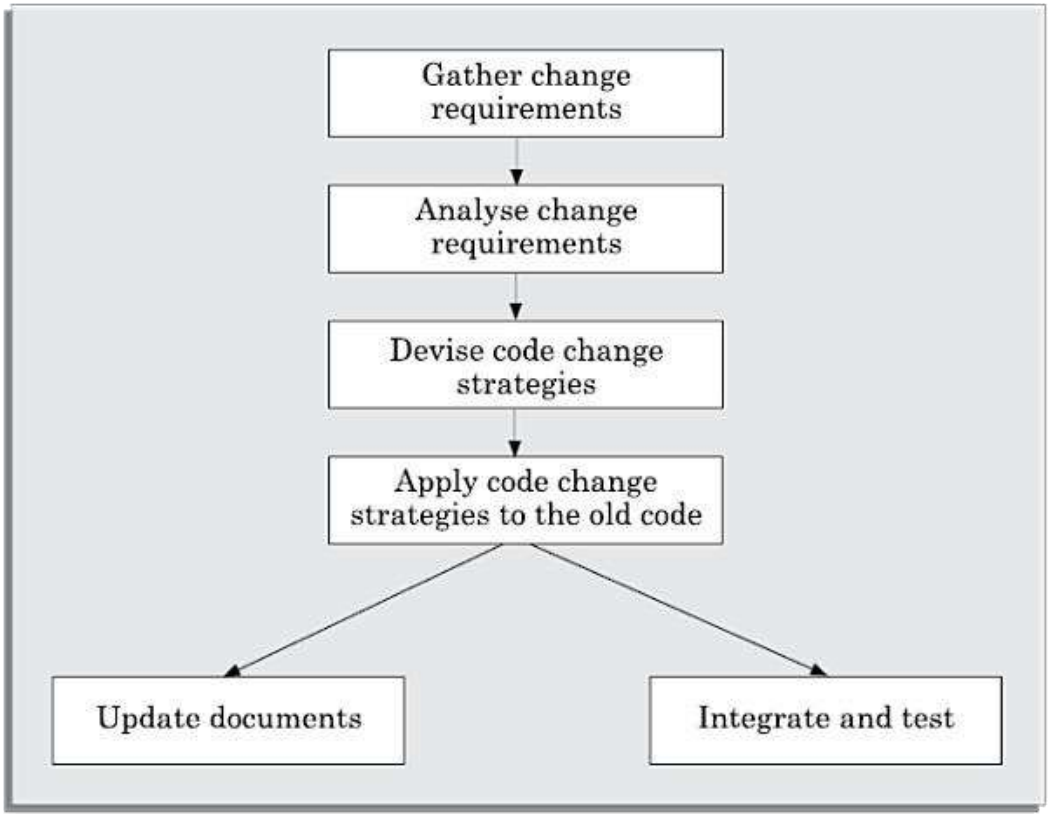


Figure 13.3: Maintenance process model 1.

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering. This process model is depicted in Figure 13.4.

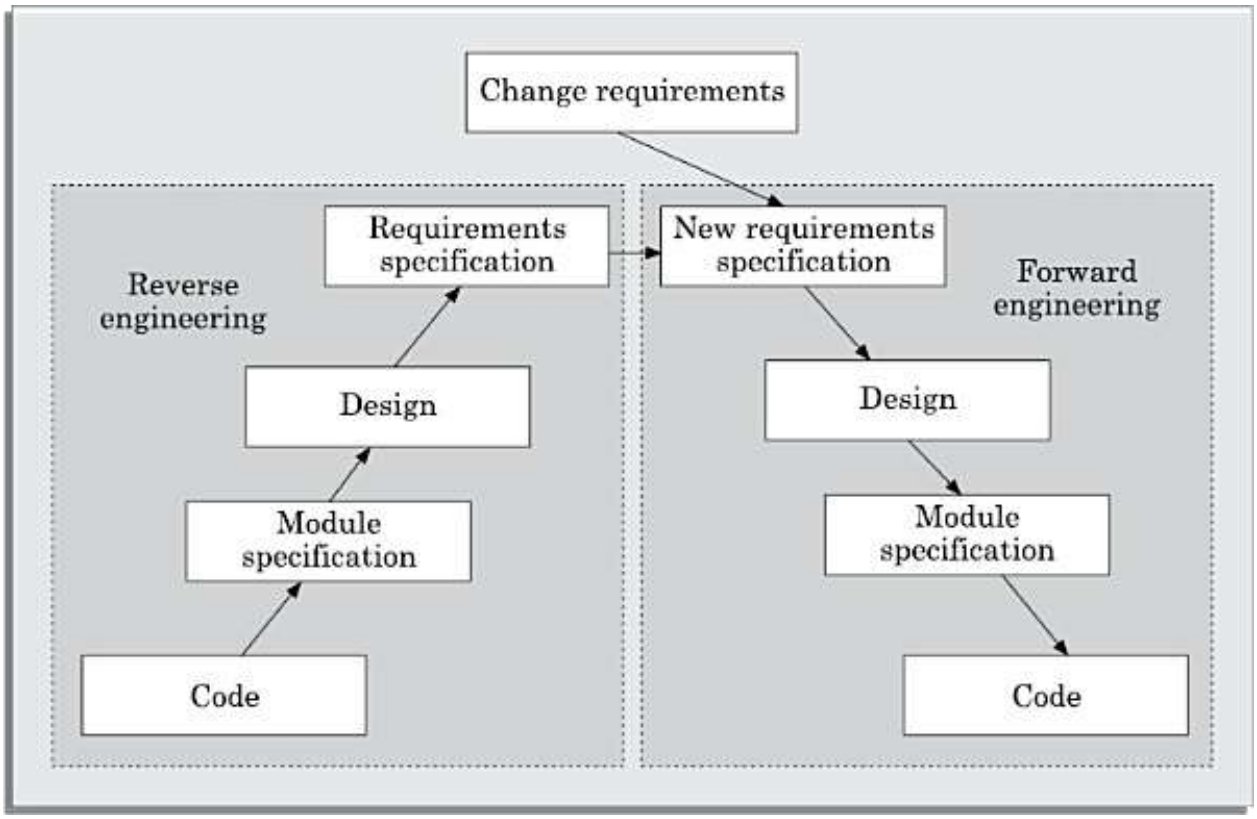


Figure 13.4: Maintenance process model 2.

The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analysed (abstracted) to extract the module specifications. The module specifications are then analysed to produce the design. The design is analysed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At this point a forward engineering is carried out to produce the new code. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products.

An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15 per cent (see Figure 13.5).

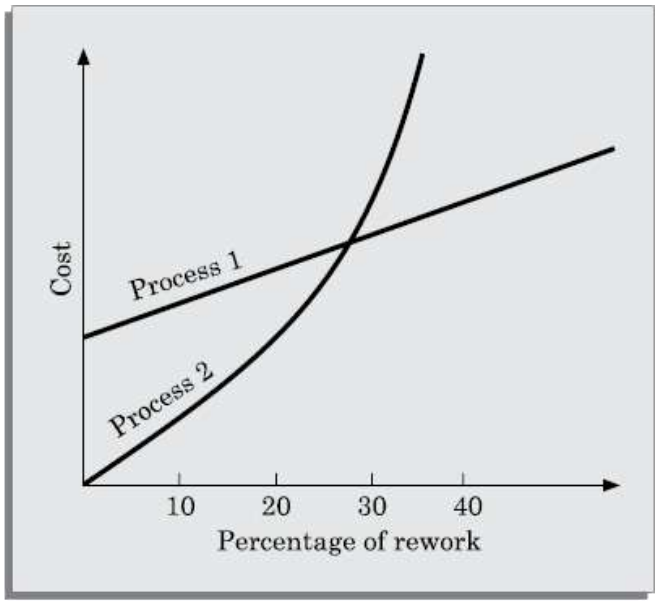


Figure 13.5: Empirical estimation of maintenance cost versus percentage rework.

Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

- Re-engineering might be preferable for products which exhibit a high failure rate.
- Re-engineering might also be preferable for legacy products having poor design and code structure.

SOFTWARE REUSE

14.1 REUSE DEFINITION?

In software engineering, **reuse** refers to the practice of using existing software components, modules, or code to build new applications or systems, rather than creating everything from scratch. This approach can significantly reduce development time, cost, and effort. Reuse can occur at various levels, such as:

Code Reuse: Reusing pre-written code modules, functions, or libraries that have been tested and optimized.

Component Reuse: Using established software components, such as pre-built classes or services, which are integrated into new applications.

Design Reuse: Reusing design patterns, architectures, or frameworks to solve common problems in new contexts.

System Reuse: Leveraging entire subsystems or platforms, like an existing content management system (CMS) or customer relationship management (CRM) system, in a new project.

The goal of software reuse is to increase productivity, improve software quality, and reduce the effort involved in the development and maintenance of software systems.

14.2 BASIC ISSUES IN ANY REUSE PROGRAM

The following are some of the basic issues that must be clearly understood for starting any reuse program:

- Component creation.
- Component indexing and storing.
- Component search.

- Component understanding.
- Component adaptation.
- Repository maintenance.

Component creation: For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. In Section 14.4, we discuss domain analysis as a promising technique which can be used to create reusable components.

Component indexing and storing

Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse. The components need to be stored in a *relational database management system* (RDBMS) or an *object-oriented database system* (ODBMS) for efficient access when the number of components becomes large.

Component searching

The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

Component understanding

The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

Component adaptation

Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

Repository maintenance

A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become

obsolete. In this case, the obsolete components might have to be removed from the repository.

14.3 A REUSE APPROACH

A promising approach that is being adopted by many organisations is to introduce a building block approach into the software development process. For this, the reusable components need to be identified after every development project is completed. The reusability of the identified components has to be enhanced and these have to be cataloged into a component library. It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components. Domain analysis is a promising approach to identify reusable components. In the following subsections, we discuss the domain analysis approach to create reusable components.

14.3.1 Domain Analysis

The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse domain

A reuse domain is a technically related set of application areas. A body of information is considered to be a problem domain for reuse, if a deep and comprehensive relationship exists among the information items as characterised by patterns of similarity among the development components of the software product. A reuse domain is a shared understanding of some community, characterised by concepts, techniques, and terminologies that show some coherence.

Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.

Just to become familiar with the vocabulary of a domain requires months of interaction with the experts. Often, one needs to be familiar with a network of related domains for successfully carrying out domain analysis. Domain analysis identifies the objects, operations, and the relationships among them.

During domain analysis, a specific community of software developers get together to discuss community-wide solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of the reusable components for a domain is called *domain*

engineering.

Evolution of a reuse domain

The ultimate results of domain analysis is development of problem-oriented languages. The problem-oriented languages are also known as *application generators*. These application generators, once developed form application development standards. The domains slowly develop. As a domain develops, we may distinguish the various stages it undergoes:

Stage 1 : There is no clear and consistent set of notations. Obviously, no reusable components are available. All software is written from scratch.

Stage 2 : Here, only experience from similar projects are used in a development effort. This means that there is only knowledge reuse.

Stage 3: At this stage, the domain is ripe for reuse. The set of concepts are stabilised and the notations standardised. Standard solutions to standard problems are available. There is both knowledge and component reuse.

Stage 4 : The domain has been fully explored. The software development for the domain can largely be automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an *application generator*.

14.3.2 Component Classification

Components need to be properly classified in order to develop an effective indexing and storage scheme. We have already remarked that hardware reuse has been very successful. If we look at the classification of hardware components for clue, then we can observe that hardware components are classified using a multilevel hierarchy. At the lowest level, the components are described in several forms—natural language description, logic schema, timing information, etc. The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's classification scheme

Each component is best described using a number of different characteristics or facets. For example, objects can be classified using the following:

- Actions they embody.
- Objects they manipulate.
- Data structures used.
- Systems they are part of, etc.

Prieto-Diaz's faceted classification scheme requires choosing an n-tuple that best fits a component. Faceted classification has advantages over enumerative classification. Strictly enumerative schemes use a pre-defined hierarchy. Therefore, these force you to search for an item that best fits the component to be classified. This makes it very difficult to search a required component. Though cross referencing to other items can be included, the resulting network becomes complicated.

14.3.3 Searching

The domain repository may contain thousands of reuse items. In such large domains, what is the most efficient way to search an item that one is looking for? A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results would do a browsing using the links provided to look up related items. The approximate automated search locates products that appear to fulfill some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository.

The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword- to-keyword, keyword-to-product, and product- to-product links. These links help to locate additional products and compare their detailed attributes. Finding a satisfactory item from the repository may require several iterations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences. However, we must remember that the items to be searched may be components, designs, models, requirements, and even knowledge.

14.3.4 Repository Maintenance

Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. Also, the links relating the different items may need to be modified to improve the effectiveness of search. The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements. On the other hand, restricting reuse to highly mature components,can sacrifice potential reuse opportunity. Making a product available before it has been thoroughly assessed can be counter productive. Negative experiences tend to dissolve the trust in the entire reuse framework.

14.3.5 Reuse without Modifications

.Once standard solutions emerge, no modifications to the program parts may be necessary. One can directly plug in the parts to develop his application. Reuse without modification is much more useful than the classical program libraries. These can be supported by compilers through linkage to run-time support routines (application generators).

Application generators translate specifications into application programs. The specification usually is written using 4GL. The specification might also be in a visual form. The programmer would create a graphical drawing using some standard available symbols. Defining what is variant and what is invariant corresponds to parameterising a subroutine to make it reusable. A subroutine's parameters are variants because the programmer can specify them while calling the subroutine. Parts of a subroutine that are not parameterised, cannot be changed.

Application generators have been applied successfully to data processing application, user interface, and compiler development. Application generators are less successful with the development of applications with close interaction with hardware such as real-time systems.

14.4 REUSE AT ORGANISATION LEVEL

Reusability should be a standard part in all software development activities including specification, design, implementation, test, etc. Ideally, there should be a steady flow of reusable components. In

practice, however, things are not so simple.

Extracting reusable components from projects that were completed in the past presents an important difficulty not encountered while extracting a reusable component from an ongoing project—typically, the original developers are no longer available for consultation. Development of new systems leads to an assortment of products, since reusability ranges from items whose reusability is immediate to those items whose reusability is highly improbable.

Achieving organisation-level reuse requires adoption of the following steps:

- Assess of an item's potential for reuse.
- Refine the item for greater reusability.
- Enter the product in the reuse repository.

In the following subsections, we elaborate these three steps required to achieve organisation- level reuse.

Assessing a product's potential for reuse

Assessment of a components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to assess a component's reusability. The programmers working in similar application domain can be used to answer the questionnaire about the product's reusability. Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored. A sample questionnaire to assess a component's reusability is the following:

- Is the component's functionality required for implementation of systems in the future?
- How common is the component's function within its domain?
- Would there be a duplication of functions within the domain if the component is taken up?
- Is the component hardware dependent?
- Is the design of the component optimised enough?
- If the component is non-reusable, then can it be decomposed to yield some reusable components?
- Can we parametrise a non-reusable component so that it becomes reusable?

Refining products for greater reusability

For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localised using data encapsulation techniques. The following refinements may be carried out:

Name generalisation: The names should be general, rather than being directly related to a specific application.

Operation generalisation: Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.

Exception generalisation: This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.

Handling portability problems: Programs typically make some assumption regarding the representation of information in the underlying machine. These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be the same on all machines. Also, programs use some function libraries, which may not be available on all host machines. A portability solution to overcome these problems is shown in Figure 14.1. The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-related calls should be routed through the portability interface. One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.

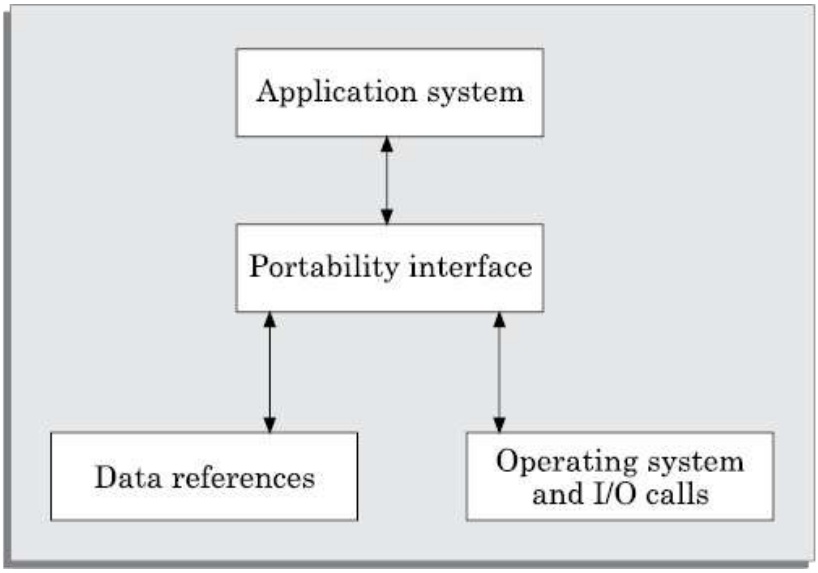


Figure 14.1: Improving reusability of a component by using a portability interface.

14.4.1 Current State of Reuse

In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organisations that most of the factors inhibiting an effective reuse program are non-technical. Some of these factors are the following:

- Need for commitment from the top management.
- Adequate documentation to support reuse.
- Adequate incentive to reward those who reuse. Both the people contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
- Providing access to and information about reusable components. Organisations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.