ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)
Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.
Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.
A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.
Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Computer Science and Engineering

Academic Year 2023-24

III B.Tech. I Semester

Software Engineering

(20APC0519)

Prepared By
Smt. G. Bhavana, *M.Tech.*
Assistant Professor
Department of CSE, AITS
bhavanagyarampalli@gmail.com

UNIT-1

INTRODUCTION

Software is a program or set of programs containing instructions that provide desired functionality. And Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.

Software Engineering is the process of designing, developing, testing, and maintaining software. It is a systematic and disciplined approach to software development that aims to create high-quality, reliable, and maintainable software. Software engineering includes a variety of techniques, tools, and methodologies, including requirements analysis, design, testing, and maintenance.

Evolution of an Art into an Engineering Discipline: Software engineering principles have evolved over the past six decades, thanks to the contributions of researchers and software professionals. It has transformed from an art to a craft and ultimately into an engineering discipline.

Early programmers employed ad hoc programming styles, now referred to as exploratory, build and fix, and code and fix approaches. The build and fix style involved quickly developing a program without any formal specifications or design, only addressing noticed imperfections later on.

Exploratory programming, an informal style, allowed programmers to develop techniques based on their intuition, experience, whims, and fancies. However, this style often resulted in poor quality, unmaintainable code, and expensive, time-consuming development.

In the early computing history, the build and fix style was widely adopted. The exploratory style was akin to an art, guided mostly by intuition, and some exceptionally skilled programmers could create elegant and correct programs using this approach.

Evolution Pattern for Engineering Disciplines:

Over the past sixty years, the evolution of software development styles has followed a familiar pattern observed in other engineering disciplines. Initially, software development was considered an esoteric art form, much like iron making or paper making in ancient times. The knowledge was limited to a select few, passed down as closely-guarded secrets from generation to generation.

Gradually, software development transitioned into a craft form, where skilled tradesmen shared their knowledge with apprentices, expanding the collective pool of expertise. Eventually, through systematic organization, documentation, and incorporation of scientific principles, software engineering emerged as a disciplined field.

In the early days of programming, there were discernible differences between good and bad programmers, with the former possessing certain principles or tricks for writing excellent programs. As time went on, these principles were organized into a cohesive body of knowledge, shaping the discipline of software engineering as we know it today.

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

Individuals, such as students working on classroom assignments or hobbyists pursuing personal projects, often develop toy software. These programs tend to be small in size and offer limited functionalities. Typically, the author of the program is the sole user and responsible for maintaining the code.

As this toy software lack extensive user-interface design and proper documentation, they may suffer from poor maintainability, efficiency, and reliability. Due to the absence of supporting documents like users' manuals, maintenance manuals, design documents, and test documents, we refer to these creations as "programs."

In contrast, professional software caters to multiple users, leading to the inclusion of robust user-interface design, comprehensive users' manuals, and extensive documentation support. With a large user base, professional software undergoes systematic design, careful implementation, and thorough testing.

A professionally developed software product encompasses not only the program code but also various associated documents like requirements specification, design documents, test documents, and users' manuals. Furthermore, these software projects are often too large and complex to be handled by a single individual, necessitating the collaborative efforts of a team of developers working together.

A professional software is developed by a group of software developers working together in a team. It is therefore necessary for them to use some systematic development methodology. Otherwise, they would find it very difficult to interface and understand each other’s work, and produce a coherent set of documents. Even though software engineering principles are primarily intended for use in development of professional software, many results of software engineering can effectively be used for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles is often not worthwhile. An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. CAR Hoare [1994] observed that rigorously using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castles for children to play.

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: Various software products, like Microsoft's Windows and Office suite, Oracle DBMS, and software bundled with camcorders or laser printers, are well-known examples. These are considered generic software products, readily available for purchase and used by a diverse range of customers. Many users essentially use the same software, making them suitable for off-the-shelf purchase.

When a software development company aims to create a generic product, it identifies features or functionalities that would be beneficial to a large user base. Based on this, the development team creates the product specification, sometimes considering feedback from numerous users. Typically, each software product is targeted at specific market segments.

Companies often find it advantageous to develop product lines catering to slightly different market segments while using variations of essentially the same software. For instance, Microsoft targets desktops and laptops with its Windows 8 operating system, high-end mobile handsets with its Windows mobile operating system, and servers with 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, if an academic institution needs software to automate student registration, grading, and fee collection, a company would develop a customized product for that specific purpose. This process involves adapting one of the company's existing software products, which might have been previously developed for another academic institution or client.

EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT

We've previously explored the concept of the exploratory program development style, which entails an informal approach to programming. In this style, programmers rely on their intuition rather than adhering strictly to the established body of knowledge within the realm of software engineering. The exploratory development style grants programmers the freedom to select the methods they deem appropriate for software development.

While this style doesn't impose specific regulations, a typical development process commences with an initial briefing from the customer. With this briefing as a foundation, developers initiate the coding phase to create a functional program. Subsequently, the software undergoes testing, during which any identified bugs are rectified. This iterative process of testing and bug fixing persists until the software meets the customer's satisfaction.

A visual representation of this approach, resembling a build and fix methodology, is depicted in Figure 1.3. Notably, the coding phase commences after an initial briefing from the customer, outlining the project's requirements. Upon completing the program development, a cycle of testing and refinement ensues until the program aligns with the customer's expectations.

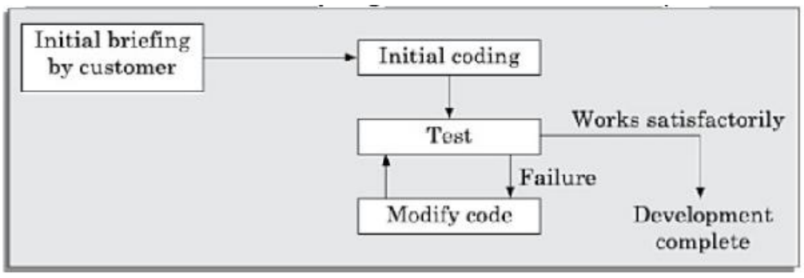


Figure 1.3: Exploratory program development.

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.

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.

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

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

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.

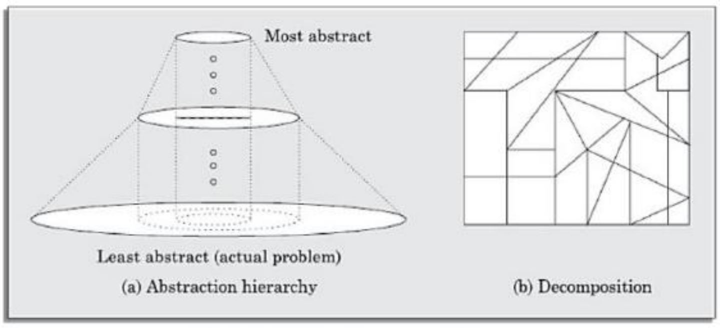


Figure 1.6: Schematic representation.

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).

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 everyday life, we use the principle of abstraction frequently to understand a problem or to assess a situation. Consider the following two examples.

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

- Consider the following situation. Suppose you are asked to develop an understanding of all the living beings inhabiting the earth. If you use the naive approach, you would start taking up one living being after another who inhabit the earth and start understanding them. Even after putting in tremendous effort, you would make little progress and left confused since there are billions of living things on earth and the information would be just too much for anyone 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.

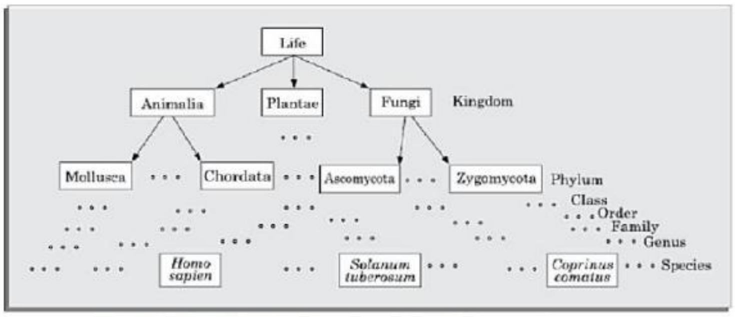


Figure 1.7: An abstraction hierarchy classifying living organisms.

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

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

Why study software engineering?

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

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

Besides the above two important skills, you would also be learning the techniques of software requirements specification user interface development, quality assurance, testing, project management, maintenance, etc. As we had already mentioned, small programs can also be written without using software engineering principles. However even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity and at the same time enable you to produce better quality programs.

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.

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.

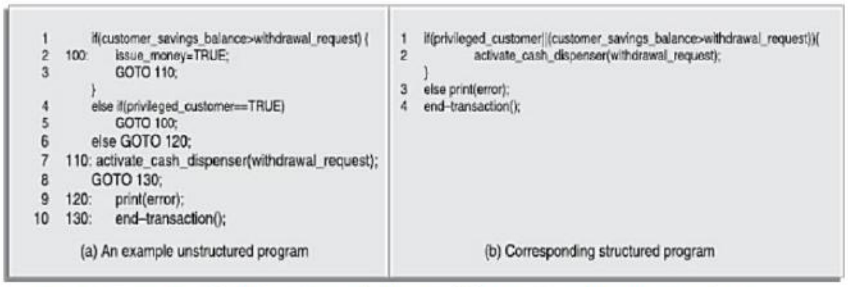
High-level Language Programming

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

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. Even today, the 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. Figure 1.8 illustrates two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.8 are drawn in Figure 1.9. Observe that the control flow structure of the program segment in Figure 1.9(b) is much simpler than that of Figure 1.9(a). By examining the code, it can be seen that Figure 1.9(a) is much harder to understand as compared to Figure 1.9(b). This example corroborates the fact that if the flow chart representation is simple, then the corresponding code should be simple. You can draw the flow chart representations of several other problems to convince yourself that a program with complex flow chart representation is indeed more difficult to understand and maintain.



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

(a) An example unstructured program

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

(b) Corresponding structured program

Figure 1.8: An example of (a) Unstructured program (b) Corresponding structured program.

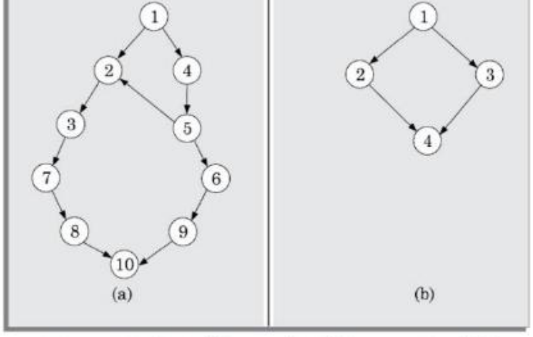


Figure 1.9: Control flow graphs of the programs of Figures 1.8(a) and (b).

Let us now try to understand why a program having good control flow structure would be easier to develop and understand. In other words, let us understand why a program with a complex flow chart representation is difficult to understand? The main reason behind this situation is that normally one understands a program by mentally tracing its execution sequence (i.e. statement sequences) to understand how the output is produced from the input values. That is, we can start from a statement producing an output, and trace back the statements in the program and understand how they produce the output by transforming the input data. Alternatively, we may start with the input data and check by running through the program how each statement processes (transforms) the input data until the output is produced. For example, for the program of Fig 1.9(a) you would have to understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and 1- 4-5-2-3-7-8-10. A program having a messy control flow (i.e. flow chart) structure, would have a large number of execution paths (see Figure 1.10). Consequently, it would become extremely difficult to determine all the execution paths, and tracing the execution sequence along all the paths trying to understand them can be nightmarish. It is therefore evident that a program having a messy flow chart representation would indeed be difficult to understand and debug.

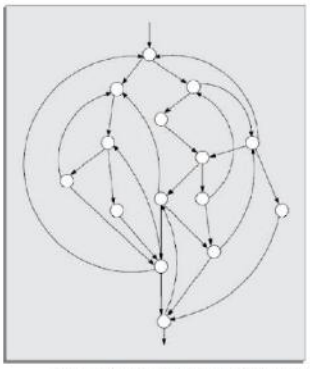


Figure 1.10: CFG of a program having too many GO TO statements.

Data Structure-oriented

Design Computers became even more powerful with the advent of integrated circuits (ICs) in the early seventies. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. 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.

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

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

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

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

DFDs: A crucial program representation for procedural program design

DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, Figure 1.11 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 as fitting of wheels, fitting the engine, spray painting the car, etc. As the partially assembled program moves along the assembly line, different workstations perform their respective jobs on the partially assembled software. Each circle in the DFD model of Figure 1.11 represents a workstation (called a process or bubble). Each workstation consumes certain input items and produces certain output items. As a car under assembly arrives at a workstation, it fetches the necessary items to be fitted from the corresponding stores (represented by two parallel horizontal lines), and as soon as the fitting work is complete passes on to the next workstation. It is easy to understand the DFD model of the car assembly plant shown in Figure 1.11 even without knowing anything regarding DFDs. In this regard, we can say that a major advantage of the DFDs is their simplicity.

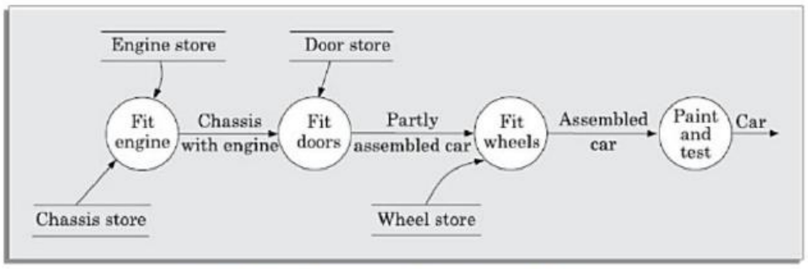


Figure 1.11: Data flow model of a car assembly plant.

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.

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.

- A significant distinction lies in the fact that the exploratory software development approach is centred on error correction (build and fix), whereas software engineering methods prioritize error prevention. Software engineering principles recognize the cost-effectiveness of preventing errors, rather than correcting them later. Even when mistakes occur during development, software engineering emphasizes error detection mainly during final product testing. In contrast, modern software development follows well-defined stages such as requirements specification, design, coding, and testing. Efforts are focused on identifying and rectifying errors within the same phase in which they arise.
- In the exploratory approach, coding equated to software development. This approach aimed to quickly build a functional system and progressively modify it until it met requirements. Exploratory programmers would dive into coding before fully grasping the problem, often leading to challenges. This approach proved costly for complex problems and generated hard-to-maintain programs. Even minor changes later became difficult. In modern software development, coding is just a fraction of the process. Other activities like design and testing demand significant effort, acknowledging that coding is a small part of the overall development activities.
- Considerable focus is directed toward requirements specifications in modern software development. Substantial effort is invested in creating an accurate and clear problem specification before commencing development activities. If the requirements specification fails to accurately capture customer needs, extensive rework may become necessary later. This rework leads to increased development costs 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.
- 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.
- By visibility we mean production of good quality, consistent and peer reviewed documents at the end of every software development activity.
- 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 smoother.
- Presently, projects undergo meticulous planning to ensure seamless development activities and resource availability. Project planning encompasses estimating, scheduling resources, and creating tracking plans. Various techniques and automation tools, such as configuration management and scheduling tools, are employed for efficient 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.

COMPUTER SYSTEMS ENGINEERING

Throughout our discussions, we've assumed the software is designed for general-purpose hardware like computers or servers. However, in various scenarios, specialized hardware is necessary for software execution. Instances include robots, factory automation, and cell phones. Cell phones, for instance, possess unique processors and devices like speakers. These systems exclusively run tailored programs. Creating such systems involves both software and hardware development, a realm known as computer systems engineering. Systems engineering encompasses software engineering while addressing the development of systems requiring dedicated hardware and software integration.

The general model of systems engineering is shown schematically in Figure 1.13. 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 hand, functionalities implemented in software is easier to extend. Further, it is difficult to implement complex functions in hardware. Also, functions implemented in hardware incur extra space, weight, manufacturing cost, and power overhead.

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 throughout the duration of system development as shown in Figure 1.13.

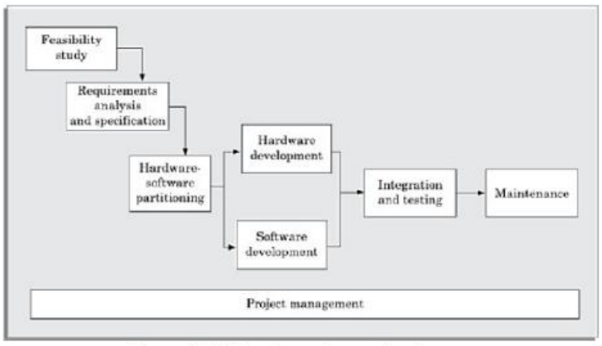


Figure 1.13: Computer systems engineering.

SOFTWARE LIFE CYCLE MODELS

A FEW BASIC CONCEPTS

In this section, we present a few basic concepts concerning the life cycle models.

Software life cycle

It is well known that all living organisms undergo a life cycle. For example, when a seed is planted, it germinates, grows into a full tree, and finally dies. Based on this concept of a biological life cycle, the term software life cycle has been defined to imply the different stages (or phases) over which a software evolves from an initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.

As we have already pointed out, the life cycle of every software starts with a request for it by one or more customers. At this stage, the customers are usually not clear about all the features that would be needed, neither can they completely describe the identified features in concrete terms, and can only vaguely describe what is needed. This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the inception stage. Starting with the inception stage, a software evolves through a series of identifiable stages (also called phases) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.

Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called maintenance) phase. As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several improvements and modifications to the software. Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user. The operation phase is usually the longest of all phases and constitutes the useful life of a software. Finally, the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of a new software having improved features and working, changed computing platforms, etc. This forms the essence of the life cycle of every software. Based on this description, we can define the software life cycle as follows:

The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.

With this knowledge of a software life cycle, we discuss the concept of a software life cycle model and explore why it is necessary to follow a life cycle model in professional software development environments.

Software development life cycle (SDLC) model

In any systematic software development scenario, certain well-defined activities need to be performed by the development team and possibly by the customers as well, for the software to evolve from one stage in its life cycle to the next. For example, for a software to evolve from the requirements specification stage to the design stage, the developers need to elicit requirements from the customers, analyse those requirements, and formally document the requirements in the form of an SRS document. A software development life cycle (SDLC) model (also called software life cycle model and software development process model) describes the different activities that need to be carried out for the software to evolve in its life cycle. Throughout our discussion, we shall use the terms software development life cycle (SDLC) and software development process interchangeably. However, some authors distinguish an SDLC from a software development process. In their usage, a software development process describes the life cycle activities more precisely and elaborately, as compared to an SDLC. Also, a development process may not only describe various activities that are carried out over the life cycle, but also prescribe a specific methodology to carry out the activities, and also recommends the the specific documents and other artifacts that should be produced at the end of each phase. In this sense, the term SDLC can be considered to be a more generic term, as compared to the development process and several development processes may fit the same SDLC.

An SDLC is represented graphically by drawing various stages of the life cycle and showing the transitions among the phases. This graphical model is usually accompanied by a textual description of various activities that need to be carried out during a phase before that phase can be considered to be complete. In simple words, we can define an SDLC as follows:

An SDLC graphically depicts the different phases through which a software evolves. It is usually accompanied by a textual description of the different activities that need to be carried out during each phase.

Process versus methodology

Though the terms process a n d methodology are at time used interchangeably, there is a subtle difference between the two. First, the term process has a broader scope and addresses either all the activities taking place during software development, or certain coarse grained activities such as design (e.g. design process), testing (test process), etc. Further, a software process not only identifies the specific activities that need to be carried out, but may also prescribe certain methodology for carrying out each activity. For example, a design process may recommend that in the design stage, the high-level design activity be carried out using Hatley and Pirbhai's structured analysis and design methodology. A methodology, on the other hand, prescribes a set of steps for carrying out a specific life cycle activity. It may also include the rationale and philosophical assumptions behind the set of steps through which the activity is accomplished.

A software development process has a much broader scope as compared to a software development methodology. A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A methodology, in contrast, describes the steps to carry out only a single or at best a few individual activities.

Why use a development process?

The primary advantage of using a development process is that it encourages development of software in a systematic and disciplined manner. Adhering to a process is especially important to the development of professional software needing team effort. When software is developed by a team rather than by an individual programmer, use of a life cycle model becomes indispensable for successful completion of the project.

Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and that helps minimise the chances of time and cost overruns.

Suppose a single programmer is developing a small program. For example, a student may be developing code for a class room assignment. The student might succeed even when he does not strictly follow a specific development process and adopts a build and fix style of development. However, it is a different ball game when a professional software is being developed by a team of programmers. Let us now understand the difficulties that may arise if a team does not use any development process, and the team members are given complete freedom to develop their assigned part of the software as per their own discretion. Several types of problems may arise. We illustrate one of the problems using an example. Suppose, a software development problem has been divided into several parts and these parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part while making assumptions about the input results required from the other parts, another might decide to prepare the test documents first, and some other developer might start to carry out the design for the part assigned to him. In this case, severe problems can arise in interfacing the different parts and in managing the overall development. Therefore, ad hoc development turns out to be is a sure way to have a failed project. Believe it or not, this is exactly what has caused many project failures in the past!

When a software is developed by a team, it is necessary to have a precise understanding among the team members as to—when to do what. In the absence of such an understanding, if each member at any time would do whatever activity he feels like doing. This would be an open invitation to developmental chaos and project failure. The use of a suitable life cycle model is crucial to the successful completion of a team-based development project. But, do we need an SDLC model for developing a small program. In this context, we need to distinguish between programming-in-the-small and programming-in-the-large.

Programming-in-the-small refers to development of a toy program by a single programmer. Whereas programming-in-the-large refers to development of a professional software through team effort. While development of a software of the former type could succeed even while an individual programmer uses a build and fix style of development, use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.

Why document a development process?

It is not enough for an organisation to just have a well-defined development process, but the development process needs to be properly documented. To understand the reason for this, let us consider that a development organisation does not document its development process. In this case, its developers develop o n l y an informal understanding of the development process. An informal understanding of the development process among the team members can create several problems during development. We have identified a few important problems that may crop up when a development process is not adequately documented. Those problems are as follows:

- A well-documented process model precisely outlines each activity in the life cycle. It also describes methodologies for conducting these activities when necessary. Without proper documentation, activity sequences can become unclear, leading to confusion among different teams. For instance, code reviews might lack structure without documented methods. Loose definitions prompt subjective judgments. For instance, when to design test cases or whether to document and rigorously detail them becomes a matter of debate. Clear documentation is crucial to avoid such challenges and maintain consistency in software development processes.
- An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process. Therefore, an undocumented process serves as a hint to the developers to loosely follow the process. The symptoms of an undocumented process are easily visible—designs are shabbily done, reviews are not carried out rigorously, etc.
- A project team might often have to tailor a standard process model for use in a specific project. It is easier to tailor a documented process model, when it is required to modify certain activities or phases of the life cycle. For example, consider a project situation that requires the testing activities to be outsourced to another organisation. In this case, a documented process model would help to identify where exactly the required tailoring should occur.
- A documented process model, as we discuss later, is a mandatory requirement of the modern quality assurance standards such as ISO 9000 and SEI CMM. This means that unless a software organisation has a documented process, it would not qualify for accreditation with any of the quality standards. In the absence of a quality certification for the organisation, the customers would be suspicious of its capability of developing quality software and the organisation might find it difficult to win tenders for software development.

A documented development process forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner. A documented development process model, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process.

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.

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.

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.

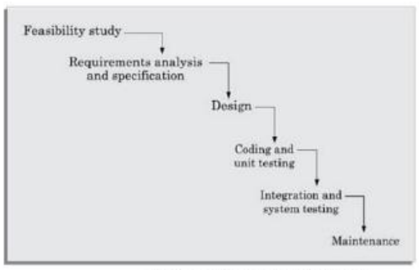

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

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. An activity that spans all phases of software development is project management. Since it spans the entire project duration, no specific phase is named after it. Project management, nevertheless, is an important activity in the life cycle and deals with managing t h e software development and maintenance activities.

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.

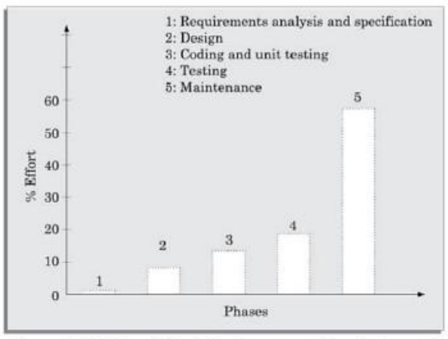


Figure 2.2: Relative effort distribution among different phases of a typical product.

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.

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.

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.
- **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 testing is carried out to verify that the interfaces among different units are working satisfactorily. On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

System testing usually consists of three different kinds of testing activities:

- **α-testing:** testing is the system testing performed by the development team.
- **β-testing:** This is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

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.

Iterative Waterfall Model

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 be 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.

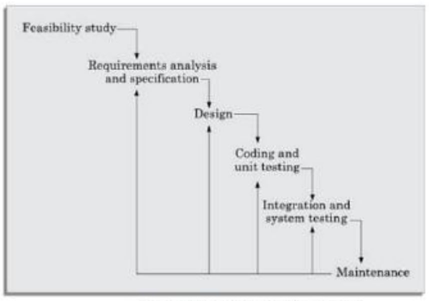

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

Figure 2.3: Iterative waterfall model.

Phase containment of errors

Programmers, despite their caution, can make mistakes during lifecycle activities, leading to errors or bugs in their work. Detecting these errors early, ideally in the same phase they occur, minimizes correction time and effort. Early identification in design prevents more complex issues during later stages like testing. Prompt error detection is cost-effective, although catching all errors in the same phase isn't always feasible. Regardless, early error detection remains crucial.

The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors.

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.

- Phase overlap is common due to task distribution among team members. If strictly sequential phases are followed, early finishers wait for others, causing inefficiency, resource wastage, and cost escalation. To avoid this, in actual projects, phases overlap. Developers move to the next phase after completing their tasks, instead of waiting for all teammates to finish.

Considering these situations, the effort distribution for different phases with time would be as shown in Figure 2.4.

Shortcomings of the iterative waterfall model The iterative waterfall model is a simple and intuitive software development model. It was used satisfactorily during 1970s and 1980s. However, the characteristics of software development projects have changed drastically over years. In the 1970s and 1960s, software development projects spanned several years and mostly involved generic software product development. The projects are now shorter, and involve Customised software development. Further, software was earlier developed from scratch. Now the emphasis is on as much reuse of code and other project artifacts as possible. Waterfall-based models have worked satisfactorily over last many years in the past. The situation has changed substantially now. As pointed out in the first chapter several decades back, every software was developed from scratch. Now, not only software has become very large and complex, very few (if at all any) software project is being developed from scratch. The software services (customised software) are poised to become the dominant types of projects. In the present software development projects, use of waterfall model causes several problems. In this context, the agile models have been proposed about a decade back that attempt to overcome the important shortcomings of the waterfall model by suggesting certain radical modification to the waterfall style of software development.

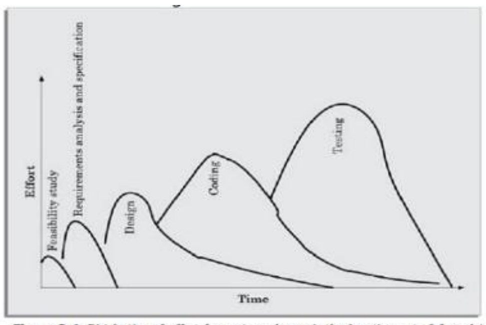


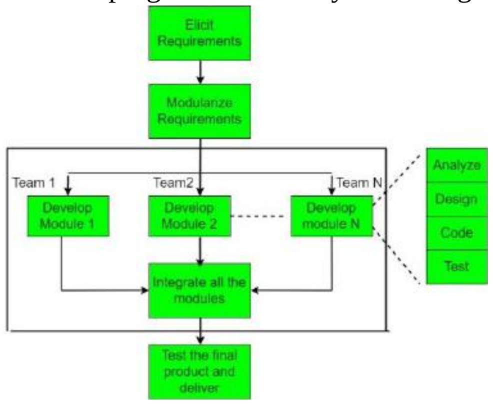
Figure 2.4: Distribution of effort for various phases in the iterative waterfall model.

The Rapid Application Development Model (RAD)

The Rapid Application Development Model was first proposed by IBM in the 1980s. The RAD model is a type of incremental process model in which there is extremely short development cycle. When the requirements are fully understood and the component-based construction approach is adopted then the RAD model is used. Various phases in RAD are Requirements Gathering, Analysis and Planning, Design, Build or Construction, and finally Deployment.

The critical feature of this model is the use of powerful development tools and techniques. A software project can be implemented using this model if the project can be broken down into small modules wherein each module can be assigned independently to separate teams. These modules can finally be combined to form the final product. Development of each module involves the various basic steps as in the waterfall model i.e. analyzing, designing, coding, and then testing, etc. as shown in the figure. Another striking feature of this model is a short time span i.e. the time frame for delivery(time-box) is generally 60-90 days.

Multiple teams work on developing the software system using RAD model parallely.



The use of powerful developer tools such as JAVA, C++, Visual BASIC, XML, etc. is also an integral part of the projects. This model consists of 4 basic phases:

1. **Requirements Planning –** It involves the use of various techniques used in requirements elicitation like brainstorming, task analysis, form analysis, user scenarios, FAST (Facilitated Application Development Technique), etc. It also consists of the entire structured plan describing the critical data, methods to obtain it, and then processing it to form a final refined model.
2. **User Description –** This phase consists of taking user feedback and building the prototype using developer tools. In other words, it includes re-examination and validation of the data collected in the first phase. The dataset attributes are also identified and elucidated in this phase.
3. **Construction –** In this phase, refinement of the prototype and delivery takes place. It includes the actual use of powerful automated tools to transform processes and data models into the final working product. All the required modifications and enhancements are too done in this phase.
4. **Cutover –** All the interfaces between the independent modules developed by separate teams have to be tested properly. The use of powerfully automated tools and subparts makes testing easier. This is followed by acceptance testing by the user.

The process involves building a rapid prototype, delivering it to the customer, and taking feedback. After validation by the customer, the SRS document is developed and the design is finalized.

When to use RAD Model?

When the customer has well-known requirements, the user is involved throughout the life cycle, the project can be time-boxed, the functionality delivered in increments, high performance is not required, low technical risks are involved and the system can be modularized. In these cases, we can use the RAD Model. when it is necessary to design a system that can be divided into smaller units within two to three months. when there is enough money in the budget to pay for both the expense of automated tools for code creation and designers for modeling.

Advantages:

- The use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at the initial stages.
- Reduced costs as fewer developers are required.
- The use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.

- It is easier to accommodate changing requirements due to the short iteration time spans.
- Productivity may be quickly boosted with a lower number of employees.

Disadvantages:

- The use of powerful and efficient tools requires highly skilled professionals.
- The absence of reusable components can lead to the failure of the project.
- The team leader must work closely with the developers and customers to close the project on time.
- The systems which cannot be modularized suitably cannot use this model.
- Customer involvement is required throughout the life cycle.
- It is not meant for small-scale projects as in such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.
- Not every application can be used with RAD.

Applications:

1. This model should be used for a system with known requirements and requiring a short development time.
2. It is also suitable for projects where requirements can be modularized and reusable components are also available for development.
3. The model can also be used when already existing system components can be used in developing a new system with minimum changes.
4. This model can only be used if the teams consist of domain experts. This is because relevant knowledge and the ability to use powerful techniques are a necessity.
5. The model should be chosen when the budget permits the use of automated tools and techniques required.

Drawbacks of rapid application development:

- It requires multiple teams or a large number of people to work on the scalable projects.
- This model requires heavily committed developer and customers. If commitment is lacking then RAD projects will fail.
- The projects using RAD model requires heavy resources.
- If there is no appropriate modularization then RAD projects fail. Performance can be problem to such projects.
- The projects using RAD model find it difficult to adopt new technologies.

Agile Development Models

In earlier days, the Iterative Waterfall Model was very popular for completing a project. But nowadays, developers face various problems while using it to develop software. The main difficulties included handling customer change requests during project development and the high cost and time required to incorporate these changes. To overcome these drawbacks of the Waterfall Model, in the mid-1990s the Agile Software Development model was proposed.

The Agile Model was primarily designed to help a project adapt quickly to change requests. So, the main aim of the Agile model is to facilitate quick project completion. To accomplish this task, agility is required. Agility is achieved by fitting the process to the project and removing activities that may not be essential for a specific project. Also, anything that is a waste of time and effort is avoided.

The Agile Model refers to a group of development processes. These processes share some basic characteristics but do have certain subtle differences among themselves.

Agile SDLC Models/Methods

- **Models:** Crystal Agile methodology places a strong emphasis on fostering effective communication and collaboration among team members, as well as taking into account the human elements that are crucial for a successful development process. This methodology is particularly beneficial for projects with a high degree of uncertainty, where requirements tend to change frequently.

- **Atern:** This methodology is tailored for projects with moderate to high uncertainty where requirements are prone to change frequently. Its clear-cut roles and responsibilities focus on delivering working software in short time frames. Governance practices set it apart and make it an effective approach for teams and projects.

- **Feature-driven development:** This approach is implemented by utilizing a series of techniques, like creating feature lists, conducting model evaluations, and implementing a design-by-feature method, to meet its goal. This methodology is particularly effective in ensuring that the end product is delivered on time and that it aligns with the requirements of the customer.

- **Scrum:** This methodology serves as a framework for tackling complex projects and ensuring their successful completion. It is led by a Scrum Master, who oversees the process, and a Product Owner, who establishes the priorities. The Development Team, accountable for delivering the software, is another key player.

- **Extreme programming (XP):** It uses specific practices like pair programming, continuous integration, and test-driven development to achieve these goals. Extreme programming is ideal for projects that have high levels of uncertainty and require frequent changes, as it allows for quick adaptation to new requirements and feedback.

- **Lean Development:** It is rooted in the principles of lean manufacturing and aims to streamline the process by identifying and removing unnecessary steps and activities. This is achieved through practices such as continuous improvement, visual management, and value stream mapping, which helps in identifying areas of improvement and implementing changes accordingly.

- **Unified Process:** Unified Process is a methodology that can be tailored to the specific needs of any given project. It combines elements of both waterfall and Agile methodologies, allowing for an iterative and incremental approach to development. This means that the UP is characterized by a series of iterations, each of which results in a working product increment, allowing for continuous improvement and the delivery of value to the customer.

All Agile methodologies discussed above share the same core values and principles, but they may differ in their implementation and specific practices. Agile development requires a high degree of collaboration and communication among team members, as well as a willingness to adapt to changing requirements and feedback from customers.

In the Agile model, the requirements are decomposed into many small parts that can be incrementally developed. The Agile model adopts Iterative development. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and can be completed within a couple of weeks only. At a time one iteration is planned, developed, and deployed to the customers. Long-term plans are not made.

Steps in the Agile Model

The agile model is a combination of iterative and incremental process models. The steps involve in agile SDLC models are:

- Requirement gathering
- Design the Requirements
- Construction / Iteration
- Testing / Quality Assurance

- Deployment
- Feedback

Steps in Agile Model

1. Requirement Gathering: In this step, the development team must gather the requirements, by interaction with the customer. development team should plan the time and effort needed to build the project. Based on this information you can evaluate technical and economic feasibility.

2. Design the Requirements: In this step, the development team will use user-flow-diagram or high-level UML diagrams to show the working of the new features and show how they will apply to the existing software. Wire framing and designing user interfaces are done in this phase.

3. Construction / Iteration: In this step, development team members start working on their project, which aims to deploy a working product.

4. Testing / Quality Assurance: Testing involves Unit Testing, Integration Testing, and System Testing**.** A brief introduction of these three tests is as follows:

5. Unit Testing: Unit testing is the process of checking small pieces of code to ensure that the individual parts of a program work properly on their own. Unit testing is used to test individual blocks (units) of code.

- **Integration Testing:** Integration testing is used to identify and resolve any issues that may arise when different units of the software are combined.

- **System Testing:** Goal is to ensure that the software meets the requirements of the users and that it works correctly in all possible scenarios.

5. Deployment: In this step, the development team will deploy the working project to end users.

6. Feedback: This is the last step of the **Agile Model.** In this, the team receives feedback about the product and works on correcting bugs based on feedback provided by the customer.

The time required to complete an iteration is known as a Time Box. Time-box refers to the maximum amount of time needed to deliver an iteration to customers. So, the end date for an iteration does not change. However, the development team can decide to reduce the delivered functionality during a Time-box if necessary to deliver it on time. The Agile model's central principle is delivering an increment to the customer after each Time-box.

Principles of the Agile Model

- To establish close contact with the customer during development and to gain a clear understanding of various requirements, each Agile project usually includes a customer representative on the team. At the end of each iteration stakeholders and the customer representative review, the progress made and re-evaluate the requirements.
- The agile model relies on working software deployment rather than comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer representative in intervals of a few weeks.
- Requirement change requests from the customer are encouraged and efficiently incorporated.
- It emphasizes having efficient team members and enhancing communications among them is given more importance. It is realized that improved communication among the development team members can be achieved through face-to-face communication rather than through the exchange of formal documents.

- It is recommended that the development team size should be kept small (5 to 9 people) to help the team members meaningfully engage in face-to-face communication and have a collaborative work environment.
- The agile development process usually deploys Pair Programming. In Pair programming, two programmers work together at one workstation. One does coding while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so.

Characteristics of the Agile Process

- Agile processes must be adaptable to technical and environmental changes. That means if any technological changes occur, then the agile process must accommodate them.
- The development of agile processes must be incremental. That means, in each development, the increment should contain some functionality that can be tested and verified by the customer.
- The customer feedback must be used to create the next increment of the process.
- The software increment must be delivered in a short span of time.
- It must be iterative so that each increment can be evaluated regularly.

When To Use the Agile Model?

- When frequent modifications need to be made, this method is implemented.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with the team all the time.
- when the project needs to be delivered quickly.
- Projects with few regulatory requirements or not certain requirements.
- projects utilizing a less-than-strict current methodology
- Those undertakings where the product proprietor is easily reachable
- Flexible project schedules and budgets.

Advantages of the Agile Model

- Working through Pair programming produces well-written compact programs which have fewer errors as compared to programmers working alone.
- It reduces the total development time of the whole project.
- Agile development emphasizes face-to-face communication among team members, leading to better collaboration and understanding of project goals.
- Customer representatives get the idea of updated software products after each iteration. So, it is easy for him to change any requirement if needed.
- Agile development puts the customer at the center of the development process, ensuring that the end product meets their needs.

Disadvantages of the Agile Model

- The lack of formal documents creates confusion and important decisions taken during different phases can be misinterpreted at any time by different team members.
- It is not suitable for handling complex dependencies.
- The agile model depends highly on customer interactions so if the customer is not clear, then the development team can be driven in the wrong direction.

- Agile development models often involve working in short sprints, which can make it difficult to plan and forecast project timelines and deliverables. This can lead to delays in the project and can make it difficult to accurately estimate the costs and resources needed for the project.
- Agile development models require a high degree of expertise from team members, as they need to be able to adapt to changing requirements and work in an iterative environment. This can be challenging for teams that are not experienced in agile development practices and can lead to delays and difficulties in the project.
- Due to the absence of proper documentation, when the project completes and the developers are assigned to another project, maintenance of the developed project can become a problem.

SPIRAL MODEL

The Spiral Model is one of the most important Software Development Life Cycle models, which provides support for Risk Handling. In its diagrammatic representation, it looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. Each loop of the spiral is called a Phase of the software development process.

The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, the project manager has an important role to develop a product using the spiral model.

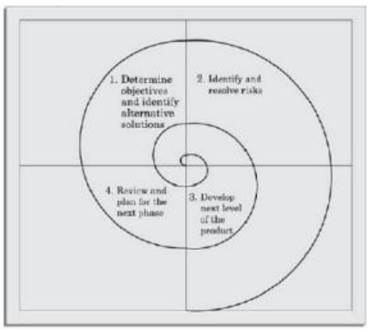
The Spiral Model is a **Software Development Life Cycle (SDLC)** model that provides a systematic and iterative approach to software development. It is based on the idea of a spiral, with each iteration of the spiral representing a complete software development cycle, from requirements gathering and analysis to design, implementation, testing, and maintenance.

What Are the Phases of Spiral Model?

The Spiral Model is a risk-driven model, meaning that the focus is on managing risk through multiple iterations of the software development process. It consists of the following phases:

- **Planning:** The first phase of the Spiral Model is the planning phase, where the scope of the project is determined and a plan is created for the next iteration of the spiral.
- **Risk Analysis:** In the risk analysis phase, the risks associated with the project are identified and evaluated.
- **Engineering:** In the engineering phase, the software is developed based on the requirements gathered in the previous iteration.
- **Evaluation:** In the evaluation phase, the software is evaluated to determine if it meets the customer’s requirements and if it is of high quality.
- **Planning:** The next iteration of the spiral begins with a new planning phase, based on the results of the evaluation.
- The Spiral Model is often used for complex and large software development projects, as it allows for a more flexible and adaptable approach to software development. It is also well-suited to projects with significant uncertainty or high levels of risk.

The Radius of the spiral at any point represents the expenses(cost) of the project so far, and the angular dimension represents the progress made so far in the current phase.


1. Determine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for the next phase

Each phase of the Spiral Model is divided into four quadrants as shown in the above figure. The functions of these four quadrants are discussed below-

- **Objectives determination and identify alternative solutions:** Requirements are gathered from the customers and the objectives are identified, elaborated, and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.
- **Identify and resolve Risks:** During the second quadrant, all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution are identified and the risks are resolved using the best possible strategy. At the end of this quadrant, the Prototype is built for the best possible solution.
- **Develop the next version of the Product:** During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.
- **Review and plan for the next Phase:** In the fourth quadrant, the Customers evaluate the so-far developed version of the software. In the end, planning for the next phase is started.

Risk Handling in Spiral Model

A risk is any adverse situation that might affect the successful completion of a software project. The most important feature of the spiral model is handling these unknown risks after the project has started. Such risk resolutions are easier done by developing a prototype. The spiral model supports coping with risks by providing the scope to build a prototype at every phase of software development.

The Prototyping Model also supports risk handling, but the risks must be identified completely before the start of the development work of the project. But in real life, project risk may occur after the development work starts, in that case, we cannot use the Prototyping Model. In each phase of the Spiral Model, the features of the product dated and analyzed, and the risks at that point in time are identified and are resolved through prototyping. Thus, this model is much more flexible compared to other SDLC models.

Why Spiral Model is called Meta Model?

The Spiral model is called a Meta-Model because it subsumes all the other SDLC models. For example, a single loop spiral actually represents the Iterative Waterfall Model. The spiral model incorporates the stepwise approach of the Classical Waterfall Model. The spiral model uses the approach of the Prototyping Model by building a prototype at the start of each phase as a risk-handling technique. 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.

Advantages of the Spiral Model

Below are some advantages of the Spiral Model.

- **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.
- **Good for large projects:** It is recommended to use the Spiral Model in large and complex projects.
- **Flexibility in Requirements:** Change requests in the Requirements at a later phase can be incorporated accurately by using this model.
- **Customer Satisfaction:** Customers can see the development of the product at the early phase of the software development and thus, they habituated with the system by using it before completion of the total product.
- **Iterative and Incremental Approach:** The Spiral Model provides an iterative and incremental approach to software development, allowing for flexibility and adaptability in response to changing requirements or unexpected events.
- **Emphasis on Risk Management:** The Spiral Model places a strong emphasis on risk management, which helps to minimize the impact of uncertainty and risk on the software development process.
- **Improved Communication:** The Spiral Model provides for regular evaluations and reviews, which can improve communication between the customer and the development team.
- **Improved Quality:** The Spiral Model allows for multiple iterations of the software development process, which can result in improved software quality and reliability.

Disadvantages of the Spiral Model

Below are some main disadvantages of the spiral model.

- **Complex:** The Spiral Model is much more complex than other SDLC models.
- **Expensive:** Spiral Model is not suitable for small projects as it is expensive.
- **Too much dependability on Risk Analysis:** The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced experts, it is going to be a failure to develop a project using this model.
- **Difficulty in time management:** As the number of phases is unknown at the start of the project, time estimation is very difficult.
- **Complexity:** The Spiral Model can be complex, as it involves multiple iterations of the software development process.
- **Time-Consuming:** The Spiral Model can be time-consuming, as it requires multiple evaluations and reviews.
- **Resource Intensive:** The Spiral Model can be resource-intensive, as it requires a significant investment in planning, risk analysis, and evaluations.

The most serious issue we face in the cascade model is that taking a long length to finish the item, and the product became obsolete. To tackle this issue, we have another methodology, which is known as the Winding model or spiral model. The winding model is otherwise called the cyclic model.

When to Use the Spiral Model?

- When a project is vast in software engineering, a spiral model is utilized.
- A spiral approach is utilized when frequent releases are necessary.
- When it is appropriate to create a prototype
- When evaluating risks and costs is crucial
- The spiral approach is beneficial for projects with moderate to high risk.
- The SDLC's spiral model is helpful when requirements are complicated and ambiguous.
- If modifications are possible at any moment
- When committing to a long-term project is impractical owing to shifting economic priorities.

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 much 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.

Selecting an Appropriate Life Cycle Model for a Project

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.

UNIT-II

SOFTWARE PROJECT MANAGEMENT(SPM)

SPM Complexities:

Software Project Management (SPM) is a proper way of planning and leading software projects. It is a part of project management in which software projects are planned, implemented, monitored, and controlled.

Need for Software Project Management: Software is a non-physical product. Software development is a new stream in business and there is very little experience in building software products. Most of the software products are made to fit clients' requirements. The most important is that the basic technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. Such type of business and environmental constraints increase risk in software development hence it is essential to manage software projects efficiently. It is necessary for an organization to deliver quality products, keep the cost within the client's budget constrain and deliver the project as per schedule. Hence in order, software project management is necessary to incorporate user requirements along with budget and time constraints.

Software Project Management consists of Several Different Types of Management:

1. **Conflict Management:** Conflict management is the process to restrict the negative features of conflict while increasing the positive features of conflict. The goal of conflict management is to improve learning and group results including efficacy or performance in an organizational setting. Properly managed conflict can enhance group results.

2. **Risk Management:** Risk management is the analysis and identification of risks that is followed by synchronized and economical implementation of resources to minimize, operate and control the possibility or effect of unfortunate events or to maximize the realization of opportunities.

3. **Requirement Management:** It is the process of analyzing, prioritizing, tracking, and documenting requirements and then supervising change and communicating to pertinent stakeholders. It is a continuous process during a project.

4. **Change Management:** Change management is a systematic approach for dealing with the transition or transformation of an organization's goals, processes, or technologies. The purpose of change management is to execute strategies for effecting change, controlling change, and helping people to adapt to change.

5. **Software Configuration Management:** Software configuration management is the process of controlling and tracking changes in the software, part of the larger cross-disciplinary field of configuration management. Software configuration management includes revision control and the inauguration of baselines.

6. **Release Management:** Release Management is the task of planning, controlling, and scheduling the build-in deploying releases. Release management ensures that the organization delivers new and enhanced services required by the customer while protecting the integrity of existing services.

Aspects of Software Project Management:

The list of focus areas it can tackle and the broad upsides of the Software Project. Management are:

1. **Planning:** The software project manager lays out the complete project's blueprint. The project plan will outline the scope, resources, timelines, techniques, strategy, communication, testing, and maintenance steps. SPM can aid greatly here.

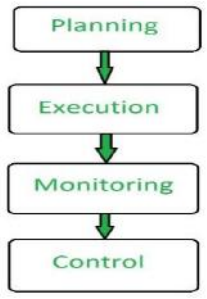
2. **Leading:** A software project manager brings together and leads a team of engineers, strategists, programmers, designers, and data scientists. Leading a team necessitates exceptional communication, interpersonal, and leadership abilities. One can only hope to do this effectively if one sticks with the core SPM principles.

3. **Execution:** SPM comes to the rescue here also as the person in charge of software projects (if well versed with SPM/Agile methodologies) will ensure that each stage of the project is completed successfully. measuring progress, monitoring to check how teams function, and generating status reports are all part of this process.

4. **Time management:** Abiding by a timeline is crucial to completing deliverables successfully. This is especially difficult when managing software projects because changes to the original project charter are unavoidable over time. To assure progress in the face of blockages or changes, software project managers ought to be specialists in managing risk and emergency preparedness. This Risk Mitigation and management is one of the core tents of the philosophy of SPM.

5. **Budget:** Software project managers, like conventional project managers, are responsible for generating a project budget and adhering to it as closely as feasible, regulating spending and reassigning funds as needed. SPM teaches us how to effectively manage the monetary aspect of projects to avoid running into a financial crunch later on in the project.

6. **Maintenance:** Software project management emphasizes continuous product testing to find and repair defects early, tailor the end product to the needs of the client, and keep the project on track. The software project manager makes ensuring that the product is thoroughly tested, analyzed, and adjusted as needed. Another point in favour of SPM.



Downsides of Software Project Management:

Numerous issues can develop if a Software project manager lacks the necessary expertise or knowledge. Software Project management has several drawbacks, including resource loss, scheduling difficulty, data protection concerns, and interpersonal conflicts between Developers/Engineers/Stakeholders. Furthermore, outsourcing work or recruiting additional personnel to complete the project may result in hefty costs for one's company.

1. **Costs are high:** Consider spending money on various kinds of project management tools, software, & services if ones engage in Software Project Management strategies. These initiatives can be expensive and time-consuming to put in place. Because your team will be using them as well, they may require training. One may need to recruit subject matter experts or specialists to assist with a project, depending on the circumstances. Stakeholders will frequently press for the inclusion of features that were not originally envisioned. All of these factors can quickly drive up a project's cost.
2. **Complexity will be increased:** Software Project management is a multi-stage, complex process. Unfortunately, some specialists might have a propensity to overcomplicate everything, which can lead to confusion among teams and lead to delays in project completion. They may also become dogmatic and specific in their ideas, resulting in a difficult work atmosphere. Projects having a larger scope are typically more arduous to complete, especially if there isn't a dedicated team committed completely to the project. Members of cross-functional teams may lag far behind their daily tasks, adding to the overall complexity of the project being worked on.
3. **Overhead in Communication:** Recruits enter your organisation when we hire software project management personnel. This provides a steady flow of communication that may or may not match a company's culture. As a result, it is advised that you maintain your crew as small as feasible. The communication overhead tends to skyrocket when a team becomes large enough. When a large team is needed for a project, it's critical to identify software project managers who can conduct effective communication with a variety of people.
4. **Lack of originality:** Software Project managers can sometimes provide little or no space for creativity. Team leaders either place an excessive amount of emphasis on management processes or impose hard deadlines on their employees, requiring them to develop and operate code within stringent guidelines. This can stifle innovative thought and innovation that could be beneficial to the project. When it comes to Software project management, knowing when to encourage creativity and when to stick to the project plan is crucial. Without Software project management personnel, an organization can perhaps build and ship code more quickly. However, employing a trained specialist to handle these areas, on the other hand, can open up new doors and help the organisation achieve its objectives more quickly and more thoroughly.

Responsibility of a software development manager:

A software project manager is the most important person inside a team who takes the overall responsibilities to manage the software projects and plays an important role in the successful completion of the projects. A project manager has to face many difficult situations to accomplish these works. In fact, the job responsibilities of a project manager range from invisible activities like building up team morale to highly visible customer presentations. Most of the managers take responsibility for writing the project proposal, 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. The task of a project manager are classified into two major types:

1. Project planning
2. Project monitoring and control

Project planning

Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirement analysis and specification phase. Once a project has been found to be feasible, Software project managers started project planning. Project planning is completed before any development phase starts. Project planning involves estimating several characteristics of a project and then plan the project activities based on these estimations. Project planning is done with most care and attention. A wrong estimation can result in schedule slippage. Schedule delay can cause customer dissatisfaction, which may lead to a project failure. Before starting a software project, it is essential to determine the tasks to be performed and properly manage allocation of tasks among individuals involved is the software development. Hence, planning is important as it results in effective software development. project planning is an organized and integrated management process, which focuses on activities required for successful completion of the project. It prevents obstacles that arise in the project such as changes in projects or organizations objectives, non- availability of resources, and so on. Project planning also helps in better utilization of resources and optimal usage of the allotted time for a project. The other objectives of project planning are listed below. It defines the roles and responsibilities of the project management team members. It ensures that the project management team works according to the business objectives. It checks feasibility of the schedule and user requirements. It determines project constraints- several individuals help in planning the project. These include senior management and project management team. For effective project planning, in addition to a very good knowledge of various estimation techniques, past experience is also very important. During the project planning the project manager performs the following activities:

1. **Project Estimation:** Project Size Estimation is the most important parameter based on which all other estimations like cost, duration and effort are made.
    - **Cost Estimation:** Total expenses to develop the software product is estimated.
    - **Time Estimation:** The total time required to complete the project.
    - **Effort Estimation:** The effort needed to complete the project is estimated.
2. **Scheduling:** After the completion of the estimation of all the project parameters, scheduling for manpower and other resources is done.
3. **Staffing:** Team structure and staffing plans are made.
4. **Risk Management:** The project manager should identify the unanticipated risks that may occur during project development risk, analyze the damage that might cause these risks, and take a risk reduction plan to cope with these risks.
5. **Miscellaneous plans:** This includes making several other plans such as quality assurance plans, configuration management plans, etc.

- **Lead the team:** The project manager must be a good leader who makes a team of different members of various skills and can complete their individual tasks.
- **Motivate the team-member:** One of the key roles of a software project manager is to encourage team members to work properly for the successful completion of the project.
- **Tracking the progress:** The project manager should keep an eye on the progress of the project. A project manager must track whether the project is going as per plan or not. If any problem arises, then take the necessary action to solve the problem. Moreover, check whether the product is developed by maintaining correct coding standards or not.
- **Liaison:** The project manager is the link between the development team and the customer. Project manager analysis the customer requirements and convey it to the development team and keep telling the progress of the project to the customer. Moreover, the project manager checks whether the project is fulfilling the customer's requirements or not.
- **Monitoring and reviewing:** Project monitoring is a continuous process that lasts the whole time a product is being developed, during which the project manager compares actual progress and cost reports with anticipated reports as soon as possible. While most firms have a formal system in place to track progress, qualified project managers may still gain a good understanding of the project's development by simply talking with participants.
- **Documenting project report:** The project manager prepares the documentation of the project for future purposes. The reports contain detailed features of the product and various techniques. These reports help to maintain and enhance the quality of the project in the future.
- **Reporting:** Reporting project status to the customer and his or her organization is the responsibility of the project manager. Additionally, they could be required to prepare brief, well-organized pieces that summarise key details from in-depth studies.

- Knowledge of project estimation techniques
- Good decision-making abilities at the right time
- Previous experience managing a similar types of projects
- Good communication skills to meet the customer satisfaction
- A project manager must encourage all the team members to successfully develop the product
- He must know the various type of risks that may occur and the solution to these problems

Metrics for Project Size Estimation:

Project size estimation is a crucial aspect of software engineering, as it helps in planning and allocating resources for the project. Here are some of the popular project size estimation techniques used in software engineering:

Expert Judgment: In this technique, a group of experts in the relevant field estimates the project size based on their experience and expertise. This technique is often used when there is limited information available about the project.

Analogous Estimation: This technique involves estimating the project size based on the similarities between the current project and previously completed projects. This technique is useful when historical data is available for similar projects.

Bottom-up Estimation: In this technique, the project is divided into smaller modules or tasks, and each task is estimated separately. The estimates are then aggregated to arrive at the overall project estimate.

Three-point Estimation: This technique involves estimating the project size using three values: optimistic, pessimistic, and most likely. These values are then used to calculate the expected project size using a formula such as the PERT formula.

Function Points: This technique involves estimating the project size based on the functionality provided by the software. Function points consider factors such as inputs, outputs, inquiries, and files to arrive at the project size estimate.

Use Case Points: This technique involves estimating the project size based on the number of use cases that the software must support. Use case points consider factors such as the complexity of each use case, the number of actors involved, and the number of use cases.

Each of these techniques has its strengths and weaknesses, and the choice of technique depends on various factors such as the project's complexity, available data, and the expertise of the team.

Estimation of the size of the software is an essential part of Software Project Management. It helps the project manager to further predict the effort and time which will be needed to build the project. Various measures are used in project size estimation. Some of these are:

- Lines of Code
- Number of entities in ER diagram
- Total number of processes in detailed data flow diagram
- Function points

1. Lines of Code (LOC): As the name suggests, LOC counts the total number of lines of source code in a project. The units of LOC are:

- KLOC- Thousand lines of code
- NLOC- Non-comment lines of code
- KDSI- Thousands of delivered source instruction

The size is estimated by comparing it with the existing systems of the same kind. The experts use it to predict the required size of various components of software and then add them to get the total size.

It's tough to estimate LOC by analyzing the problem definition. Only after the whole code has been developed can accurate LOC be estimated. This statistic is of little utility to project managers because project planning must be completed before development activity can begin.

Two separate source files having a similar number of lines may not require the same effort. A file with complicated logic would take longer to create than one with simple logic. Proper estimation may not be attainable based on LOC.

The length of time it takes to solve an issue is measured in LOC. This statistic will differ greatly from one programmer to the next. A seasoned programmer can write the same logic in fewer lines than a newbie coder.

Advantages:

- Universally accepted and is used in many models like COCOMO.
- Estimation is closer to the developer's perspective.
- Both people throughout the world utilize and accept it.
- At project completion, LOC is easily quantified.
- It has a specific connection to the result.
- Simple to use.

Disadvantages:

- Different programming languages contain a different number of lines.
- No proper industry standard exists for this technique.
- It is difficult to estimate the size using this technique in the early stages of the project.
- When platforms and languages are different, LOC cannot be used to normalize.

2. Number of entities in ER diagram: ER model provides a static view of the project. It describes the entities and their relationships. The number of entities in ER model can be used to measure the estimation of the size of the project. The number of entities depends on the size of the project. This is because more entities needed more classes/structures thus leading to more coding.

Advantages:

- Size estimation can be done during the initial stages of planning.
- The number of entities is independent of the programming technologies used.

Disadvantages:

- No fixed standards exist. Some entities contribute more to project size than others.
- Just like FPA, it is less used in the cost estimation model. Hence, it must be converted to LOC.

3. Total number of processes in detailed data flow diagram: Data Flow Diagram(DFD) represents the functional view of software. The model depicts the main processes/functions involved in software and the flow of data between them. Utilization of the number of functions in DFD to predict software size. Already existing processes of similar type are studied and used to estimate the size of the process. Sum of the estimated size of each process gives the final estimated size.

Advantages:

- It is independent of the programming language.
- Each major process can be decomposed into smaller processes. This will increase the accuracy of the estimation

Disadvantages:

- Studying similar kinds of processes to estimate size takes additional time and effort.
- All software projects are not required for the construction of DFD.

4. Function Point Analysis: In this method, the number and type of functions supported by the software are utilized to find FPC (function point count). The steps in function point analysis are:

- Count the number of functions of each proposed type.
- Compute the Unadjusted Function Points(UFP).
- Find the Total Degree of Influence(TDI).
- Compute Value Adjustment Factor(VAF).
- Find the Function Point Count(FPC).

The explanation of the above points is given below:

- **Count the number of functions of each proposed type:** Find the number of functions belonging to the following types:
    - External Inputs: Functions related to data entering the system.
    - External outputs: Functions related to data exiting the system.
    - External Inquiries: They lead to data retrieval from the system but don’t change the system.
    - Internal Files: Logical files maintained within the system. Log files are not included here.
    - External interface Files: These are logical files for other applications which are used by our system.
- **Compute the Unadjusted Function Points(UFP):** Categorise each of the five function types like simple, average, or complex based on their complexity. Multiply the count of each function type with its weighting factor and find the weighted sum. The weighting factors for each type based on their complexity are as follows:

Function type	**Simple**	**Average**	**Complex**
External Inputs	3	4	6

Function type	**Simple**	**Average**	**Complex**
External Output	4	5	7
External Inquiries	3	4	6
Internal Logical Files	7	10	15
External Interface Files	5	7	10

- **Find Total Degree of Influence:** Use the "14 general characteristics" of a system to find the degree of influence of each of them. The sum of all 14 degrees of influence will give the TDI. The range of TDI is 0 to 70. The 14 general characteristics are: Data Communications, Distributed Data Processing, Performance, Heavily Used Configuration, Transaction Rate, On-Line Data Entry, End-user Efficiency, Online Update, Complex Processing Reusability, Installation Ease, Operational Ease, Multiple Sites and Facilitate Change.
  Each of the above characteristics is evaluated on a scale of 0-5.
- **Compute Value Adjustment Factor(VAF):** Use the following formula to calculate VAF

VAF = (TDI * 0.01) + 0.65

- **Find the Function Point Count:** Use the following formula to calculate FPC

FPC = UFP * VAF

Advantages:

- It can be easily used in the early stages of project planning.
- It is independent of the programming language.
- It can be used to compare different projects even if they use different technologies (database, language, etc.).

Disadvantages:

- It is not good for real-time systems and embedded systems.
- Many cost estimation models like COCOMO use LOC and hence FPC must be converted to LOC.

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.

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.

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.

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. 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.

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.

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.

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 cannot unjustly be influenced by overly assertive and senior members.

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.

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 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 classifying 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.

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.

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.

HALSTEAD'S SOFTWARE SCIENCE—AN ANALYTICAL TECHNIQUE

Halstead's software science 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 overall 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 and 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.

STAFFING LEVEL ESTIMATION

Once the effort required to complete a software project has been estimated, the staffing requirement for the project can be determined. Putnam was the first to study the problem of determining a proper staffing pattern for software projects. He extended the classical work of Norden who had earlier investigated the staffing pattern of general research and development (R&D) type of projects. In order to appreciate the uniqueness of the staffing pattern that is desirable for software projects, we must first understand both Norden's and Putnam's results.

Norden's Work: Norden studied the staffing patterns of several R&D projects. He found that the staffing pattern of R&D type of projects is very different from that of manufacturing or sales. In a sales outlet, the number of sales staff does not usually vary with time. For example, in a supermarket the number of sales personnel would depend on the number of sales counters and would be approximately constant over time. However, the sta ffing pattern of R&D type of projects needs to change over time. At the start of an R&D project, the activities of the project are planned and initial investigations are made. During this time, the manpower requirements are low. As the project progresses, the manpower requirement increases, until it reaches a peak. Thereafter, the manpower requirement gradually diminishes.

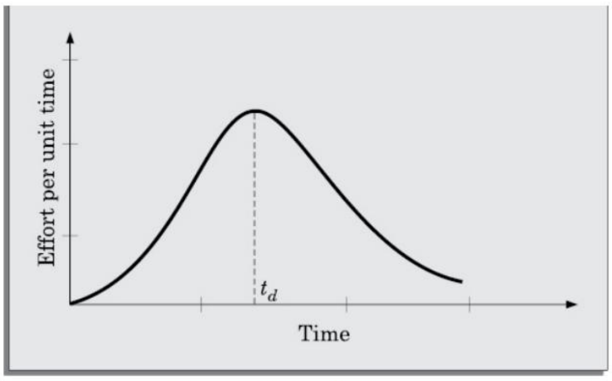


Figure 3.6: Rayleigh curve.

Norden represented the Rayleigh curve by the following equation:

$E = \frac{K}{t_d^2} * t * e^{\frac{-t^2}{2t_d^2}}$ where, E is the effort required at time t. E is an indication of the number of developers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R&D projects and were not meant to model the staffing pattern of software development projects.

Putnam's Work: Putnam studied the problem of staffing of software projects and found that the staffing pattern for software development projects has characteristics very similar to any other R&D projects. Only a small number of developers are needed at the beginning of a project to carry out the planning and specification tasks. As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product testing. After implementation and unit testing, the number of project staff falls.

Putnam found that the Rayleigh-Norden curve can be adapted to relate the number of delivered lines of code to the effort and the time required to develop the product. By analysing a large number of defence projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

where the different terms are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- $t_d$corresponds to the time of system and integration and testing. Therefore, t_d can be approximately considered as the time required to develop the software.

- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer Typical values of $C_k = 2$ for poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ for good software development environment (software engineering principles are adhered to), $C_k = 11$ for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from historical data of the organisation developing it.

For efficient resource utilisation as well as project completion over optimal duration, starting from a small number of developers, there should be a staff build-up and after a peak size has been achieved, staff reduction is required. However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve.

SCHEDULING

Scheduling the project tasks is an important project planning activity. Once a schedule has been worked out and the project gets underway, the project manager monitors the timely completion of the tasks and takes any corrective action that may be necessary whenever there is a chance of schedule slippage. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the major activities that need to be carried out to complete the project.
2. Break down each activity into tasks.
3. Determine the dependency among different tasks.
4. Establish the estimates for the time durations necessary to complete the tasks.
5. Represent the information in the form of an activity network.
6. Determine task starting and ending dates from the information represented in the activity network.
7. Determine the critical path. A critical path is a chain of tasks that determines the duration of the project.
8. Allocate resources to tasks.

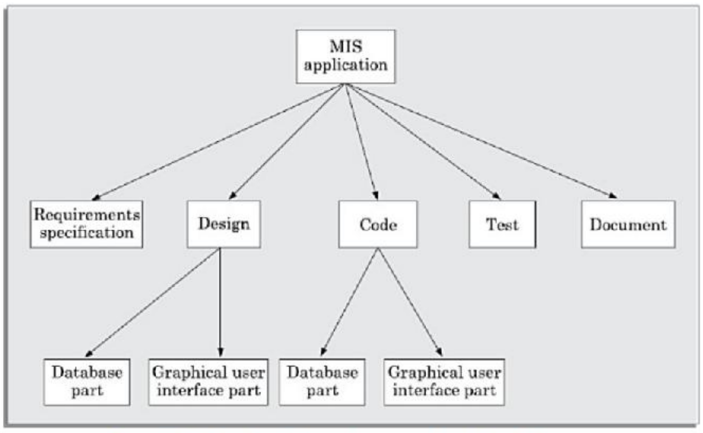
The first step in scheduling a software project involves identifying all the activities necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important activities of the project. Next, the activities are broken down into a logical set of smaller activities (sub-activities). The smallest sub-activities are called tasks which are assigned to different developers.

The smallest unit of work activities that are subject to management planning and control are called tasks.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a project evaluation and review technique (PERT) chart representation is developed. The PERT chart representation is useful to a project manager to carry out project monitoring and control. Let us now discuss the work break down structure, activity network, Gantt and PERT charts.

Work Breakdown Structure: Work breakdown structure (WBS) is used to recursively decompose a given set of activities into smaller activities. First, let us understand why it is necessary to break down project activities into tasks. Once project activities have been decomposed into a set of tasks using WBS, the time frame when each activity is to be performed is to be determined. The end of each important activity is called a milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that some milestones start getting delayed, he carefully monitors and controls the progress of the tasks, so that the overall deadline can still be met.

WBS provides a notation for representing the activities, sub-activities, and tasks needed to be carried out in order to solve a problem. Each of these is represented using a rectangle (see Figure 3.7). The root of the tree is labelled by the project name. Each node of the tree is broken down into smaller activities that are made the children of the node. To decompose an activity to a sub-activity, a good knowledge of the activity can be useful. Figure 3.7 represents the WBS of a management information system (MIS) software.


MIS application
Requirements specification
Design
Code
Test
Document
Database part
Graphical user interface part
Database part
Graphical user interface part
Figure 3.7: Work breakdown structure of an MIS problem.

Activity Networks: An activity network shows the different activities making up a project, their estimated durations, and their interdependencies. Two equivalent representations for activity networks are possible and are in use:

Activity on Node (AoN): In this representation, each activity is represented by a rectangular (some use circular) node and the duration of the activity is shown alongside each task in the node. The inter-task dependencies are shown using directional edges (see Figure 3.8).

Activity on Edge (AoE): In this representation tasks are associated with the edges. The edges are also annotated with the task duration. The nodes in the graph represent project milestones.

Activity networks were originally represented using activity on edge (AoE) representation. However, later activity on node (AoN) has become popular since this representation is easier to understand and revise.

Critical Path Method (CPM):

CPM and PERT are operation research techniques that were developed in the late 1950s. Since then, they have remained extremely popular among project managers. Of late, these two techniques have got merged and many project management tools support them as CPM/PERT. However, we point out the fundamental differences between the two and discuss CPM in this subsection and PERT in the next subsection.

A path in the activity network graph is any set of consecutive nodes and edges in this graph from the starting node to the last node. A critical path consists of a set of dependent tasks that need to be performed in a sequence and which together take the longest time to complete.

CPM is an algorithmic approach to determine the critical paths and slack times for tasks not on the critical paths involves calculating the following quantities:

Minimum time (MT): It is the minimum time required to complete the project. It is computed by determining the maximum of all paths from start to finish.

Earliest start (ES): It is the time of a task is the maximum of all paths from the start to this task. The ES for a task is the ES of the previous task plus the duration of the preceding task. **Latest start time (LST):** It is the difference between MT and the maximum of all paths from this task to the finish. The LST can be computed by subtracting the duration of the subsequent task from the LST of the subsequent task.

Earliest finish time (EF): The EF for a task is the sum of the earliest start time of the task and the duration of the task.

Latest finish (LF): LF indicates the latest time by which a task can finish without affecting the final completion time of the project. A task completing beyond its LF would cause project delay. LF of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.

Slack time (ST): The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the “flexibility” in starting and completion of tasks. ST for a task is LS-ES and can equivalently be written as LF-EF.

PERT Charts

The activity durations computed using an activity network are only estimated duration. It is therefore not possible to estimate the worst case (pessimistic) and best case (optimistic) estimations using an activity diagram. Since, the actual durations might vary from the estimated durations, the utility of the activity network diagrams are limited. The CPM can be used to determine the duration of a project, but does not provide any indication of the probability of meeting that schedule.

Project evaluation and review technique (PERT) charts are a more sophisticated form of activity chart. Project managers know that there is considerable uncertainty about how much time a task would exactly take to complete. The duration assigned to tasks by the project manager are after all only estimates. Therefore, in reality the duration of an activity is a random variable with some probability distribution. In this context, PERT charts can be used to determine the probabilistic times for reaching various project mile stones, including the final mile stone. PERT charts like activity networks consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. A PERT chart represents the statistical variations in the project estimates assuming these to be normal distribution. PERT allows for some randomness in task completion times, and therefore provides the capability to determine the probability for achieving project milestones based on the probability of completing each task along the path to that milestone. Each task is annotated with three estimates:

- **Optimistic (O):** The best possible case task completion time.

- **Most likely estimate (M):** Most likely task completion time.
- **Worst case (W):** The worst possible case task completion time.

The optimistic (O) and worst case (W) estimates represent the extremities of all possible scenarios of task completion. The most likely estimate (M) is the completion time that has the highest probability. The three estimates are used to compute the expected value of the standard deviation.

Gantt Charts

Gantt chart has been named after its developer Henry Gantt. A Gantt chart is a form of bar chart. The vertical axis lists all the tasks to be performed. The bars are drawn along the y-axis, one for each task. Gantt charts used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a unshaded part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The unshaded part shows the slack time or lax time. The lax time represents the leeway or flexibility available in meeting the latest time by which a task must be finished. Gantt charts are useful for resource planning (i.e. allocate resources to activities). The different types of resources that need to be allocated to activities include staff, hardware, and software.

Gantt chart representation of a project schedule is helpful in planning the utilisation of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different developers.

ORGANISATION AND TEAM STRUCTURES

Usually every software development organisation handles several projects at any time. Software organisations assign different teams of developers to handle different software projects. With regard to staff organisation, there are two important issues—How is the organisation as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organisations and teams can be structured. We discuss these in the following subsection.

Organisation Structure: Essentially there are three broad ways in which a software development organisation can be structured—functional format, project format, and matrix format. We discuss these three formats in the following subsection.

Functional format: In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically been shown in Figure 3.13(a). The different projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase. As a result, different teams of programmers from different functional groups perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the product evolves. Therefore, the functional format requires considerable communication among the different teams and development of good quality documents because the work of one team must be clearly understood by the subsequent teams working on the project. The functional organisation therefore mandates good quality documentation to be produced after every activity.

Project format: In the project format, the development staff are divided based on the project for which they work (See Figure 3.13(b)). A set of developers is assigned to every project at the start of the project, and remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. An advantage of the project format is that it provides job rotation. That is, every developer undertakes different life cycle activities in a project. However, it results in poor manpower utilisation, since the full project team is formed since the start of the project, and there is very little work for the team during the initial phases of the life cycle.

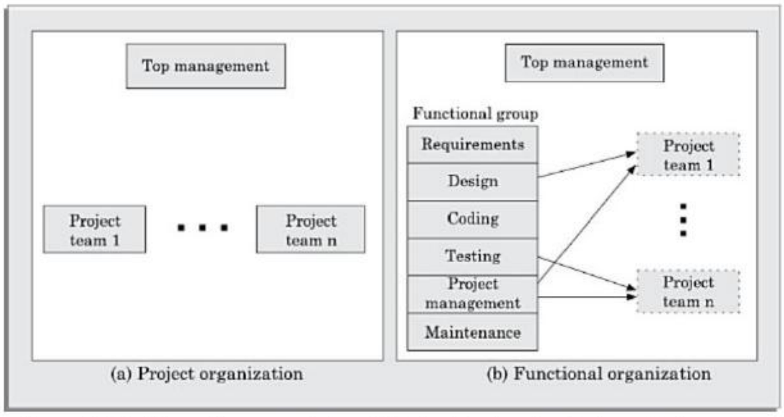


Figure 3.13: Schematic representation of the functional and project organisation.

Functional versus project formats: Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organisation are:

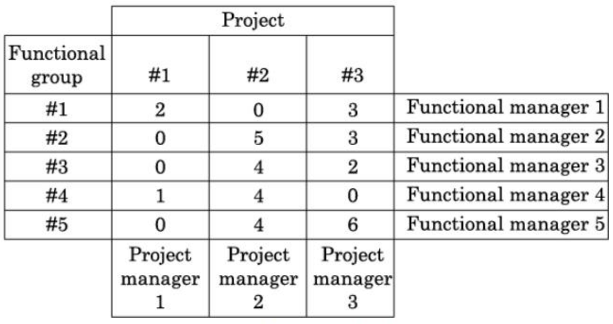
- Ease of staffing
- Production of good quality documents
- Job specialisation
- Efficient handling of the problems associated with manpower turnover.

The functional organisation allows the developers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organisation also provides an efficient solution to the staffing problem. We have already seen that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilisation of the personnel by minimizing their wait times. The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organisation. A project organisation structure forces the manager to take in almost a constant number of developers for the entire duration of his project. This results in developers idling in the initial phase of software development and are under tremendous pressure in the later phase of development. A further advantage of the functional organisation is that it is more effective in handling the problem of manpower turnover. This is because developers can be brought in from the functional pool when needed. Also, this organisation mandates production of good quality documents, so new developers can quickly get used to the work already done.

In spite of several important advantages of the functional organisation, it is not very popular in the software industry. This apparent paradox is not difficult to explain. We can easily identify the following three points:

- The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, co der, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organisations to fill slots for some roles such as the maintenance, testing, and coding groups.
- Another problem with the functional organisation is that if an organisation handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects.
- For obvious reasons the functional format is not suitable for small organisations handling just one or two projects.

Matrix format: A matrix organisation is intended to provide the advantages of both functional and project structures. In a matrix organisation, the pool of functional specialists is assigned to different projects as needed. Thus, the deployment of the different functional specialists in different projects can be represented in a matrix (see Figure 3.14). In Figure 3.14 observe that different members of a functional specialisation are assigned to different projects. Therefore, in a matrix organisation, the project manager needs to share responsibilities for the project with a number of individual functional managers.



	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Figure 3.14: Matrix organisation.

Matrix organisations can be characterised as weak or strong, depending upon the relative authority of the functional managers and the project managers. In a strong functional matrix, the functional managers have authority to assign workers to projects and project managers have to accept the assigned personnel. In a weak matrix, the project manager controls the project budget, can reject workers from functional groups, or even decide to hire outside workers.

Two important problems that a matrix organisation often suffers from are:

- Conflict between functional manager and project managers over allocation of workers.
- Frequent shifting of workers in a firefighting mode as crises occur in different projects.

Team Structure

Team structure addresses organisation of the individual project teams. Let us examine the possible ways in which the individual project teams are organised. In this text, we shall consider only three formal team structures—democratic, chief programmer, and the mixed control team organisations, although several other variations to these structures are possible. Projects of specific complexities and sizes often require specific team structures for efficient working.

In this team organisation, a senior engineer provides the technical leadership and is designated the chief programmer. The chief programmer partitions the task into many smaller tasks and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in Figure 3.15. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since the team members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer. That is, a project might suffer severely, if the chief programmer either leaves the organisation or becomes unavailable for some other reasons.

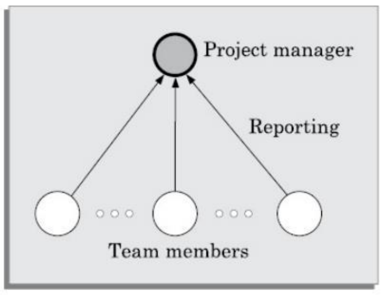


Figure 3.15: Chief programmer team structure.

Let us now try to understand the types of projects for which the chief programmer team organisation would be suitable. Suppose an organisation has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well understood problems, an organisation must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early completion outweighs other factors such as team morale, personal developments, etc.

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.

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.

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.

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:

$p = r * s$ 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.

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,

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

SOFTWARE CONFIGURATION MANAGEMENT

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g., source code, design document, SRS document, test document, user’s manual, etc. These objects are usually referred to and modified by a number of software developers throughout the life cycle of the software. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

As a software is changed, new revisions and versions get created. Before we discuss configuration management, we must be clear about the distinction between a version and a revision of a software product.

Software revision versus version: A new version of a software is created when there is significant change in functionality, technology, or the hardware it runs on, etc. On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc. Even the initial delivery might consist of several versions and more versions might be added later on. For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often systems are described as version m, release n; or simply mn. Formally, a history relation is version of can be defined between objects. This relation can be split into two sub-relations is revision of and is variant of. In the following subsections, we first discuss the necessity of configuration management and subsequently we discuss the configuration management activities and tools.

Necessity of Software Configuration Management: There are several reasons for putting an object under configuration management. The following are some of the important problems that can crop up, if configuration management is not used: every software developer has a personal copy of an object (e.g. source code). When a developer makes changes to his local copy, he is expected to intimate the changes that he has made to other developers, so that the necessary changes in interfaces could be uniformly carried out across all modules. However, not only would it eat up valuable time of the developers, but many times a developer might make changes to the interfaces in his local copies and forgets to intimate the teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the different modules are integrated, it does not work. Therefore, when several team members work on developing an application, it is necessary for them to work on a single copy of the application, otherwise inconsistencies may arise.

Problems associated with concurrent access: Possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems can appear. Assume that only a single copy of a program module is maintained, and several developers are working on it. Two developers may simultaneously carry out changes to different functions of the same module, and while saving overwrite each other. Similar problems can occur for any other deliverable object.

Providing a stable development environment: When a project work is underway, the team members need a stable environment to make progress. Suppose one developer is trying to integrate module A, with the modules B and C; since if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces recompilation of the module. When an effective configuration management is in place, the manager freezes the objects to form a baseline. A baseline is the status of all the objects under configuration control. When any of the objects under configuration control is changed, a new baseline gets formed. When any team member needs to change any of the objects under configuration control, he is provided with a copy of the baseline item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new baseline gets formed instantly. This establishes a baseline for others to use and depend on. Also, baselines may be archived periodically (archiving means copying to a safe place such as a remote storage), so that the last baseline can be recovered when there is a disaster.

System accounting and maintaining status information: System accounting denotes keeping track of who made a particular change to an object and when the change was made.

Handling variants: Existence of variants of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find that a bug exists in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, you should not have to fix it in each and every version and revision of the software separately. Making a change to one program should be reflected appropriately in all relevant versions and revisions.

Configuration Management Activities

Configuration identification: It involves deciding which parts of the system should be kept track of.

Configuration control: It ensures that changes to a system happen smoothly. Normally, a project manager performs the configuration management activity by using a configuration management tool. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the developer to change various components in a controlled manner. In the following subsections, we provide an overview of the two configuration management activities.

Configuration identification:

Project managers normally classify the objects associated with a software development into three main categories—controlled, precontrolled, and uncontrolled. Controlled objects are those that are already under configuration control. The team members must follow some formal procedures to change them. Precontrolled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not subject to configuration control. Controllable objects include both controlled and precontrolled objects. Typical controllable objects include:

- Requirements specification document
- Design documents
- Tools used to build the system, such as compilers, linkers, lexical analysers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

Configuration management plan is written during the project planning phase. It lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management increase to unreasonably high levels. On the other hand, controlling too little might lead to confusion and inconsistency when something changes.

REQUIREMENTS ANALYSIS AND SPECIFICATION

The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible. The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the software requirements specification (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather a n d analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.

How is the SRS document validated?

Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis
- Requirements specification

In the next section, we will discuss the requirements gathering and analysis activity and in the subsequent section we will discuss the requirements specification activity.

REQUIREMENTS GATHERING AND ANALYSIS

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

- Requirements gathering
- Requirements analysis

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. For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures. In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office. In this case, the analyst would have to study the input and output forms and then understand how the outputs are produced from the input data. However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult. In the absence of a working system, much more imagination and creativity is required on the part of the system analyst.

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, carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis. Given that many customers are not computer savvy; they describe their requirements very vaguely. Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete. 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 different 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.

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.

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?

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.

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 it's 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.

Project 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.

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 and not discuss the implementation issues. This view with which a requirements specification is written, has been shown in Figure 4.1. Observe that in Figure 4.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.

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 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.

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 it’s 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.

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 on 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 methods 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.

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 specification?

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.

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.

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.

EXECUTABLE SPECIFICATION AND 4GL

When the specification of a system is expressed formally or is described by using a programming language, then it becomes possible to directly execute the specification without having to design and write code for implementation. However, executable specifications are usually slow and inefficient, 4GLs (4th Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of large granularity commonality across data processing applications which have been identified and mapped to program code. 4GLs get their power from software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in 3GLs results in up to 50 per cent lower memory usage and also the program execution time can reduce up to ten folds.

Chapter 5

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.

The activities carried out during the design phase (called as design process) transform the SRS document into the design document.

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.

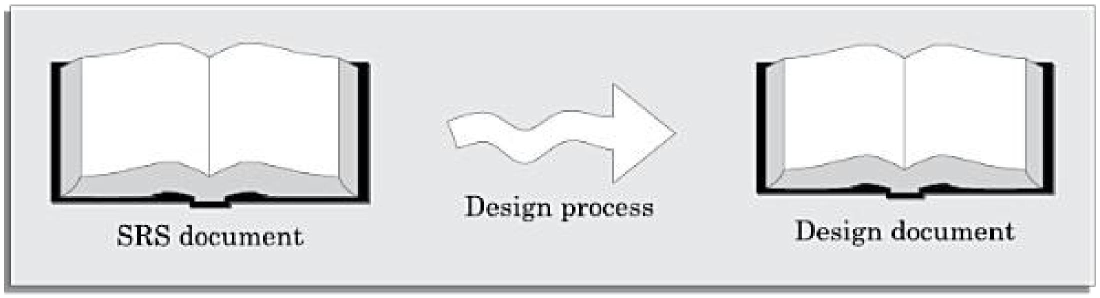


Figure 5.1: The design process.

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:

> Through high-level design, a problem is decomposed into a set of modules. The control relationships among the modules are identified, and also the interfaces among various modules are identified.

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. Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design, we confine our attention in this text to structure charts and UML diagrams only.

Once the high-level design is complete, detailed design is undertaken.

> During detailed design each module is examined carefully to design its data structures and the algorithms.

The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document. After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding. In this text, we do not discuss MSPECs and confine our attention to high-level design only.

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.

Analysis versus design

Analysis and design activities differ in goal and scope.

The goal of any analysis technique is to elaborate the customer requirements through careful thinking and at the same time consciously avoiding making any decisions regarding the exact way the system is to be implemented.

The analysis results are generic and does not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML). The analysis model would normally be very difficult to implement using a programming language.

The design model is obtained from the analysis model through transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented. The design model should be detailed enough to be easily implementable using a programming language.

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. For example, "memory size used up by a program" may be an important issue to Characterise a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations. For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. Thus for embedded applications, one may sacrifice design comprehensibility to achieve code compactness. Similarly, it is not usually true that a criterion that is crucial for some application, needs to be almost completely ignored for another application. It is therefore clear that the criteria used to judge a design solution can vary widely across different types of applications. Not only do the criteria used to judge a design solution depend on the exact application being designed, but to make the matter worse, there is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application. 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?

> Given that we are choosing from only correct design solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

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. We had already pointed out in Chapter 2 that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance. If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold. Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable. Recollect that we had already discussed in Chapter 1 that understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.

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.

It is not difficult to argue that modularity is an important characteristic of a good design solution. But, even with this, how can we compare the modularity of two alternate design solutions? From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular For example, consider two alternate design solutions to a problem that are represented in Figure 5.2, in which the modules M1 , M2 etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 5.2(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another. Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterise the modularity of a design solution based on the cohesion and coupling existing in the design.

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.

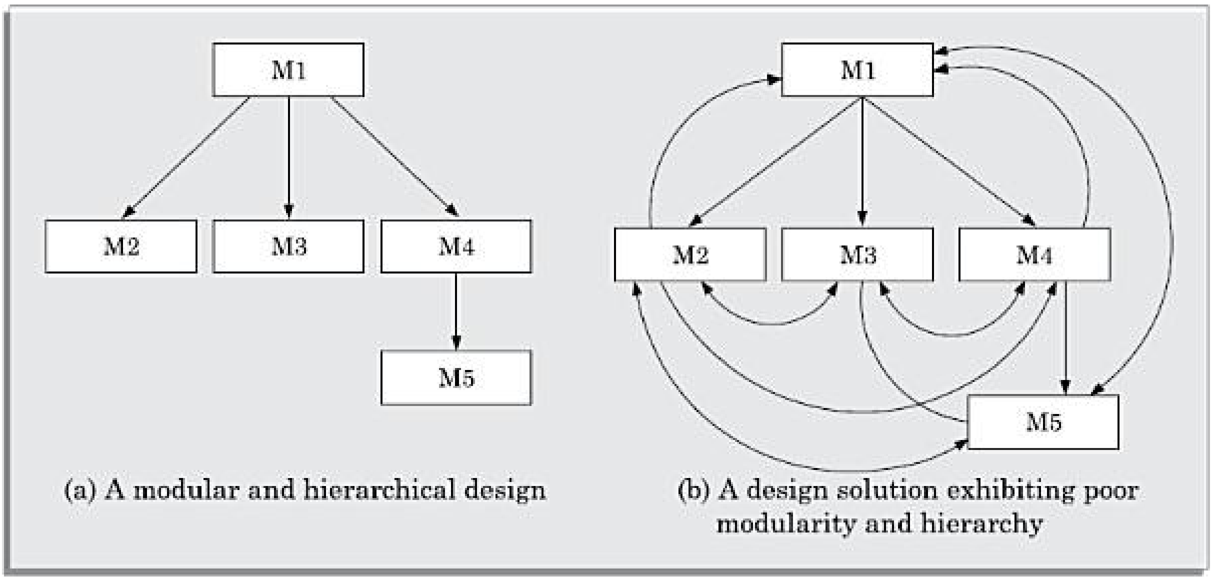

M1
M2
M3
M4
M5
(a) A modular and hierarchical design
M1
M2
M3
M4
M5
(b) A design solution exhibiting poor modularity and hierarchy

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.

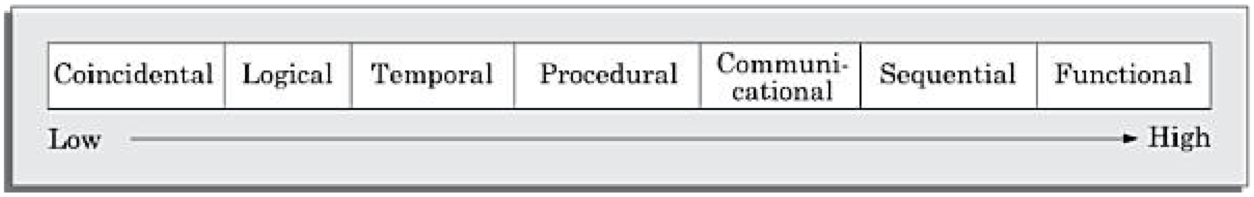


Figure 5.3: Classification of cohesion.

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 cohesion has 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.

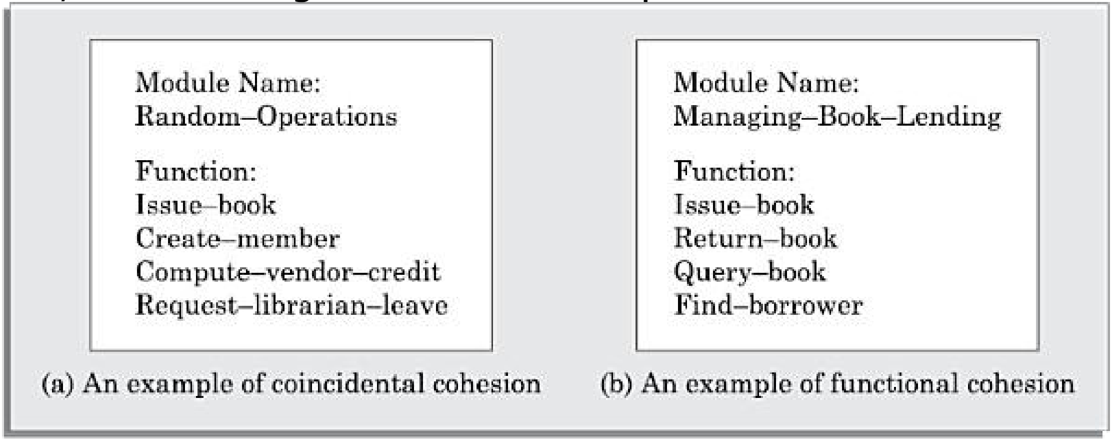


Figure 5.4: Examples of cohesion.

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."

A simple way to determine the cohesiveness of any given module is as follows. First examine what do the functions of the module perform. Then, try to write down a sentence to describe the overall work performed by the module. If you need a compound sentence to describe the functionality of the module, then it has sequential or communicational cohesion. If you need words such as "first", "next", "after", "then", etc., then it possesses sequential or temporal cohesion. If it needs words such as “initialise”, “setup”, “shut down”, etc., to define its functionality, then it has temporal cohesion.

We can now make the following observation. A cohesive module is one in which the functions interact among themselves heavily to achieve a single goal. As a result, if any of these functions is removed to a different module, the coupling would increase as the functions would now interact across two different modules.

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.

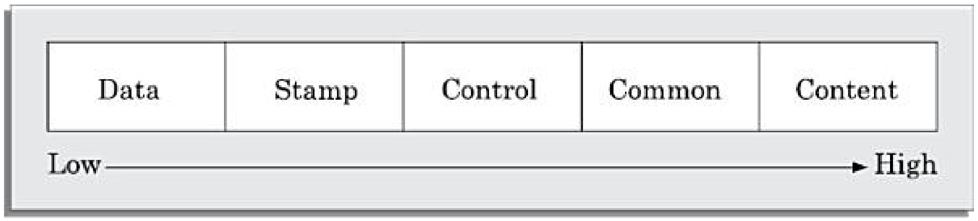


Figure 5.5: Classification of coupling.

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 LAYERED ARRANGEMENT OF MODULES

T h e control hier a r c h y represents the organisation of program components in terms of their call relationships. Thus we can say that the control hierarchy of a design is determined by the order in which different modules call each other. Many different types of notations have been used to represent the control hierarchy. The most common notation is a tree-like diagram known as a structure chart which we shall study in some detail in Chapter 6. However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used. Since, Warnier-Orr and Jackson's notations are not widely used nowadays, we shall discuss only structure charts in this text.

In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer. Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered. Observe that the design solution shown in Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer. In the following, we state the significance of a layered design and subsequently we explain it.

An important characteristic feature of a good design solution is layering of the modules. A layered design achieves control abstraction and is easier to understand and debug.

In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.

Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes). Besides, in a layered design errors are isolated, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error. Thus, debugging time reduces significantly in a layered design. On the other hand, if the different modules call each other arbitrarily, then this situation would correspond to modules arranged in a single layer. Locating an error would be both difficult and time consuming. This is because, once a failure is observed, the cause of failure (i.e. error) can potentially be in any module, and all modules would have to be investigated for the error. In the following, we discuss some important concepts and terminologies associated with a layered design:

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

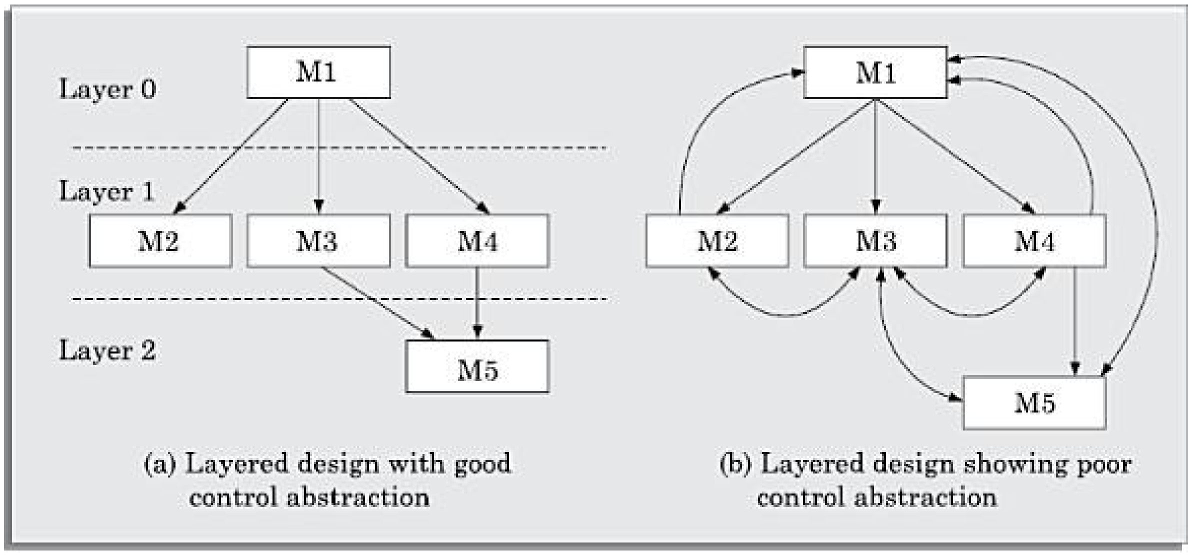

Layer 0
Layer 1
Layer 2
M1
M2
M3
M4
M5
(a) Layered design with good control abstraction
(b) Layered design showing poor control abstraction

Figure 5.6: Examples of good and poor control abstraction.

5.5 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.

5.5.1 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 member 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]

5.5.2 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.

At this point, we must emphasise that it is not necessary that an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ and Java support the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural languages—though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language. In fact, the older C++ compilers were essentially pre-processors that translated C++ code into C code.

Even though object-oriented and function-oriented techniques are remarkably different approaches to software design, yet one does not replace the other; but they complement each other in some sense. For example, usually one applies the top-down function oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to have been developed in an object-oriented fashion, but inside each class there may be a small hierarchy of functions designed in a top-down manner.

SUMMARY

- software design is typically carried out through two stages—high-level design, and detailed design. During high-level design, the important components (modules) of the system and their interactions are identified. During detailed design, the algorithms and data structures are identified.
- We discussed that there is no unique design solution to any problem and one needs to choose the best solution among a set of candidate solutions. To be able to achieve this, we identified the factors based on which a superior design can be distinguished from a inferior design.
- We discussed that understandability of a design is a major criterion determining the goodness of a design. We Characterised the understandability of design in terms of satisfactory usage of decomposition and abstraction principles. Later, we Characterised these in terms of cohesion, coupling, layering, control abstraction, fan-in, fan-out, etc.
- We identified two fundamentally different approaches to software design—function- oriented design and object-oriented design. We discussed the essential philosophy governing these two approaches and argued that these two approaches to software design are not really competing approaches but complementary approaches.

EXERCISES

1. Choose the correct option
   (a) The extent of data interchange between two modules is called:
      (i) Coupling
      (ii) Cohesion
      (iii) Structure
      (iv) Union
   (b) Which of the following type of cohesion can be considered as the strongest cohesion:
      (i) Logical
      (ii) Coincidental
      (iii) Temporal
      (iv) Functional

(c) The modules in a good software design should have which of the following characteristics:
   (i) High cohesion, low coupling
   (ii) Low cohesion, high coupling
   (iii) Low cohesion, low coupling
   (iv) High cohesion, high coupling

2. Do you agree with the following assertion? A design solution that is difficult to under- stand would lead to increased development and maintenance cost. Give reasonings for your answer.
3. What do you mean by the terms cohesion and coupling in the context of software design?
   How are these concepts useful in arriving at a good design of a system?
4. What do you mean by a modular design? How can you determine whether a given design is modular or not?
5. Enumerate the different types of cohesion that a module in a design might exhibit. Give examples of each.
6. Enumerate the different types of coupling that might exist between two modules. Give examples of each.
7. Is it true that whenever you increase the cohesion of your design, coupling in the design would automatically decrease? Justify your answer by using suitable examples.
8. What according to you are the characteristics of a good software design?
9. What do you understand by the term functional independence in the context of software design? What are the advantages of functional independence? How can functional independence in a software design be achieved?
10. Explain how the principles of abstraction and decomposition are used to arrive at a good design.
11. What do you understand by information hiding in the context of software design?
   Explain why a design approach based on the information hiding principle is likely to lead to a reusable and maintainable design. Illustrate your answer with a suitable example.
12. In the context of software development, distinguish between analysis and design with respect to intention, methodology, and the documentation technique used.
13. State whether the following statements are **TRUE** or **FALSE**. Give reasons for your answer.

(a) The essence of any good function-oriented design technique is to map the functions performing similar activities into a module.
(b) Traditional procedural design is carried out top-down whereas object-oriented design is normally carried out bottom-up.
(c) Common coupling is the worst type of coupling between two modules.
(d) Temporal cohesion is the worst type of cohesion that a module can have.
(e) The extent to which two modules depend on each other determines the cohesion of the two modules.

14. Compare relative advantages of the object-oriented and function-oriented approaches to software design.
15. Name a few well-established function-oriented software design techniques.
16. Explain the important causes of and remedies for high coupling between two software modules.
17. What problems are likely to arise if two modules have high coupling?
18. What problems are likely to occur if a module has low cohesion?
19. Distinguish between high-level and detailed designs. What documents should be produced on completion of high-level and detailed designs respectively?
20. What is meant by the term cohesion in the context of software design? Is it true that in a good design, the modules should have low cohesion? Why?
21. What is meant by the term coupling in the context of software design? Is it true that in a good design, the modules should have low coupling? Why?
22. What do you mean by modular design? What are the different factors that affect the modularity of a design? How can you assess the modularity of a design? What are the advantages of a modular design?
23. How would you improve a software design that displays very low cohesion and high coupling?
24. Explain how the overall cohesion and coupling of a design would be impacted if all modules of the design are merged into a single module.
25. Explain what do you understand by the terms decomposition and abstraction in the context of software design. How are these two principles used in arriving good procedural designs?
26. What is an ADT? What advantages accrue when a software design

technique is based on ADTs? Explain why the object paradigm is said to be based on ADTs.

27. By using suitable examples explain the following terms associated with an abstract data type (ADT)—data abstraction, data structure, data type.
28. What do you understand by the term top-down decomposition in the context of function- oriented design? Explain your answer using a suitable example.
29. What do you understand by a layered software design? What are the advantages of a layered design? Explain your answer by using suitable examples.
30. What is the principal difference between the software design methodologies based on functional abstraction and those based on data abstraction? Name at least one popular design technique based on each of these two software design paradigms.
31. What are the main advantages of using an object-oriented approach to software design over a function-oriented approach?
32. Point out three important differences between the function oriented and the object- oriented approaches to software design. Corroborate your answer through suitable examples.
33. Identify the criteria that you would use to decide which one of two alternate function- oriented design solutions to a problem is superior.
34. Explain the main differences between architectural design, high-level-design, and detailed design of a software system.

Chapter 6

FUNCTION-ORIENTED SOFTWARE DESIGN

Function-oriented design techniques were proposed nearly four decades ago. These techniques are at the present time still very popular and are currently being used in many software development organisations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software. These services provided by a software (e.g., issue book, serach book, etc., for a Library Automation Software to its users are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions.

The term top-down decomposition i s often used to denote the successive decomposition of a set of high-level functions into more detailed functions.

After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created. This module structure would possess all the characteristics of a good design identified in the last chapter.

In this text, we shall not focus on any specific design methodology. Instead, we shall discuss a methodology that has the essential features of several important function-oriented design methodologies. Such an approach shall enable us to easily assimilate any specific design methodology in the future whenever the need arises. Learning a specific methodology may become necessary for you later, since different software development houses follow different methodologies. After all, the different procedural design techniques can be considered as sister techniques that have only minor differences with respect to the methodology and notations. We shall call the design technique discussed in this text a s structured analysis/structured design (SA/SD) methodology. This technique draws heavily from the design methodologies proposed by the following authors:

- DeMarco and Yourdon [1978]
- Constantine and Yourdon [1979]
- Gane and Sarson [1979]
- Hatley and Pirbhai [1987]

The SA/SD technique can b e used to perform the high-level design of a software. The details of SA/SD technique are discussed further.

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.

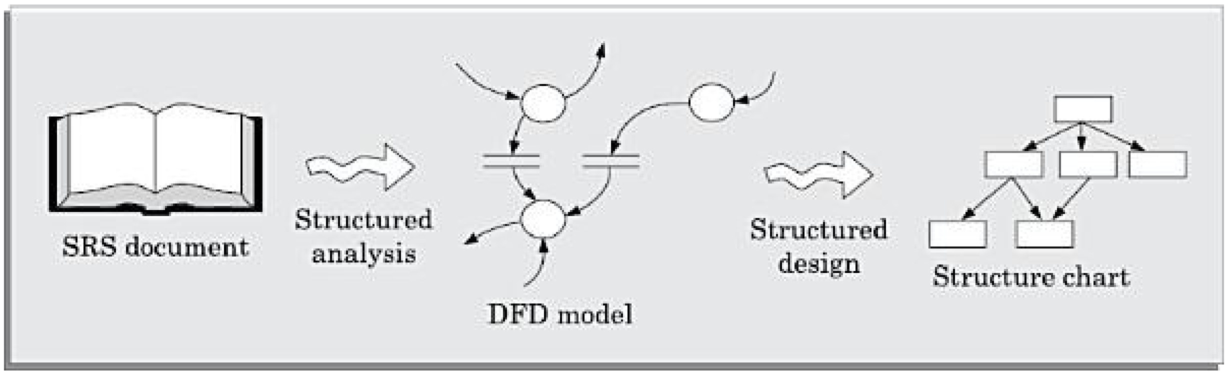


Figure 6.1: Structured analysis and structured design methodology.

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, w e 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 t h e data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. 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.

A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.

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. Recall that in Chapter 1 we had given an example (see Figure 1.10) to illustrate how a DFD can be used t o represent the processing activities and flow of material in an automated car assembling plant. We now elaborate how a DFD model can be constructed.

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:

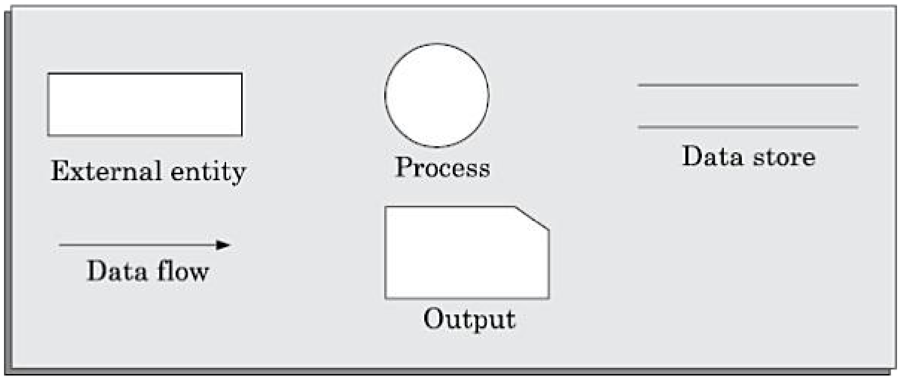


Figure 6.2: Symbols used for designing DFDs.

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 the 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 the 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.

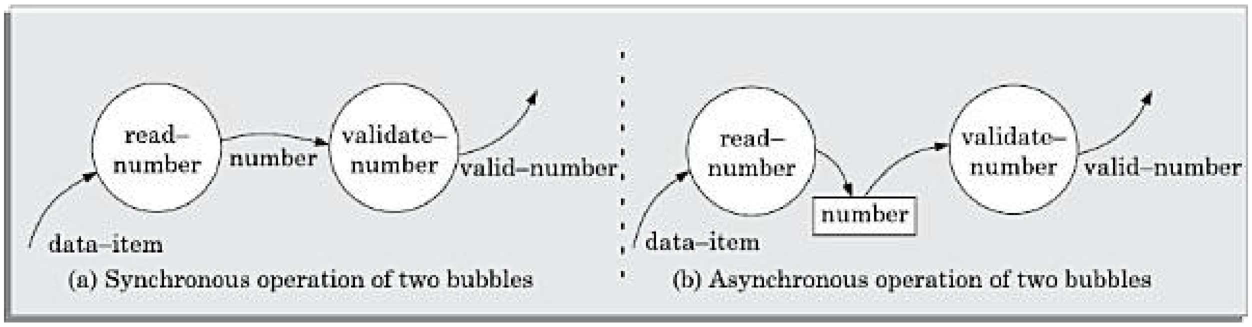


Figure 6.3: Synchronous and asynchronous data flow.

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.

$$grossP\ ay = regularP\ ay + overtimeP\ ay$$

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 difficult 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 is 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'.

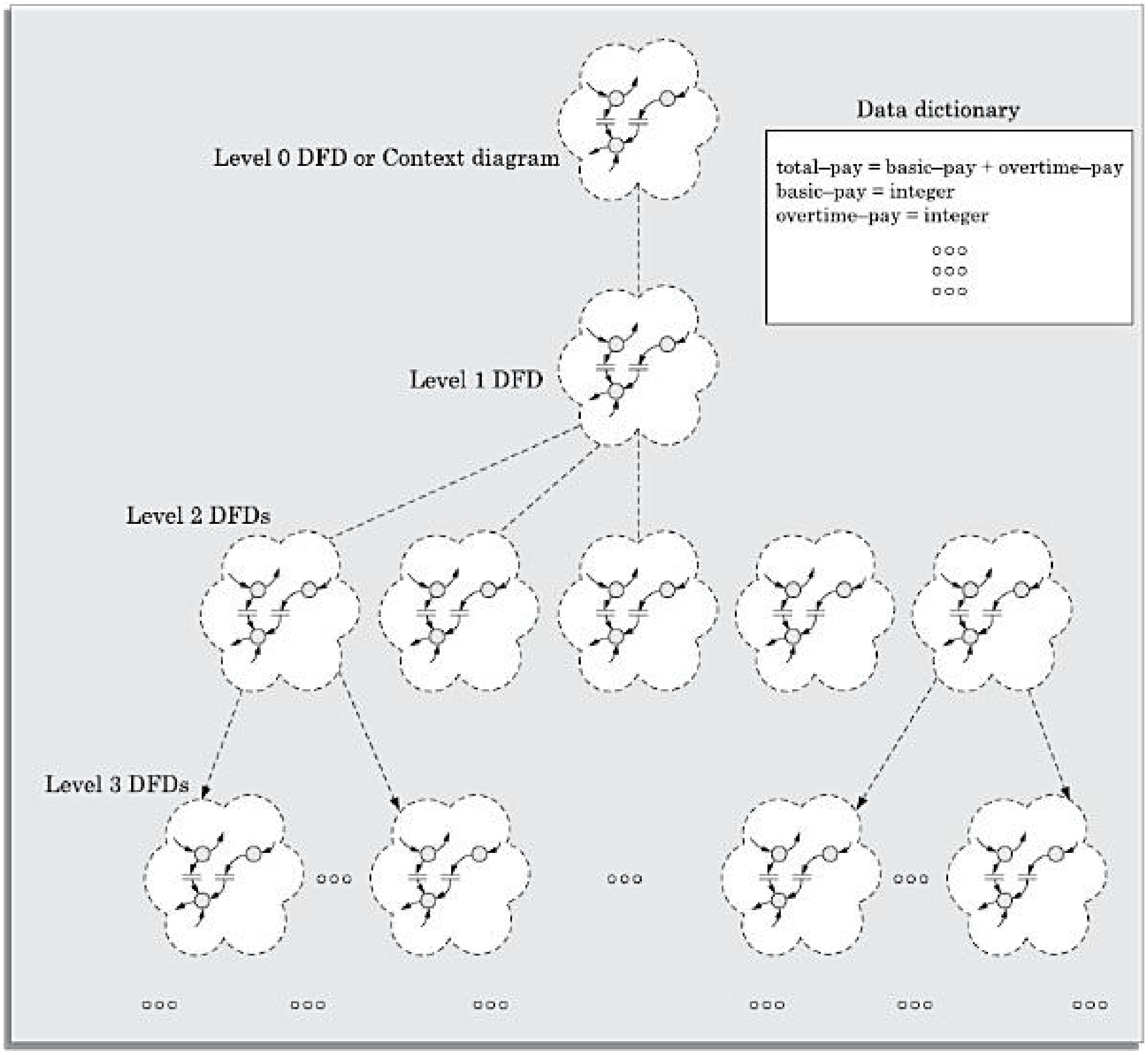


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

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 of 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 or 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 make 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 of 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 d2 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.

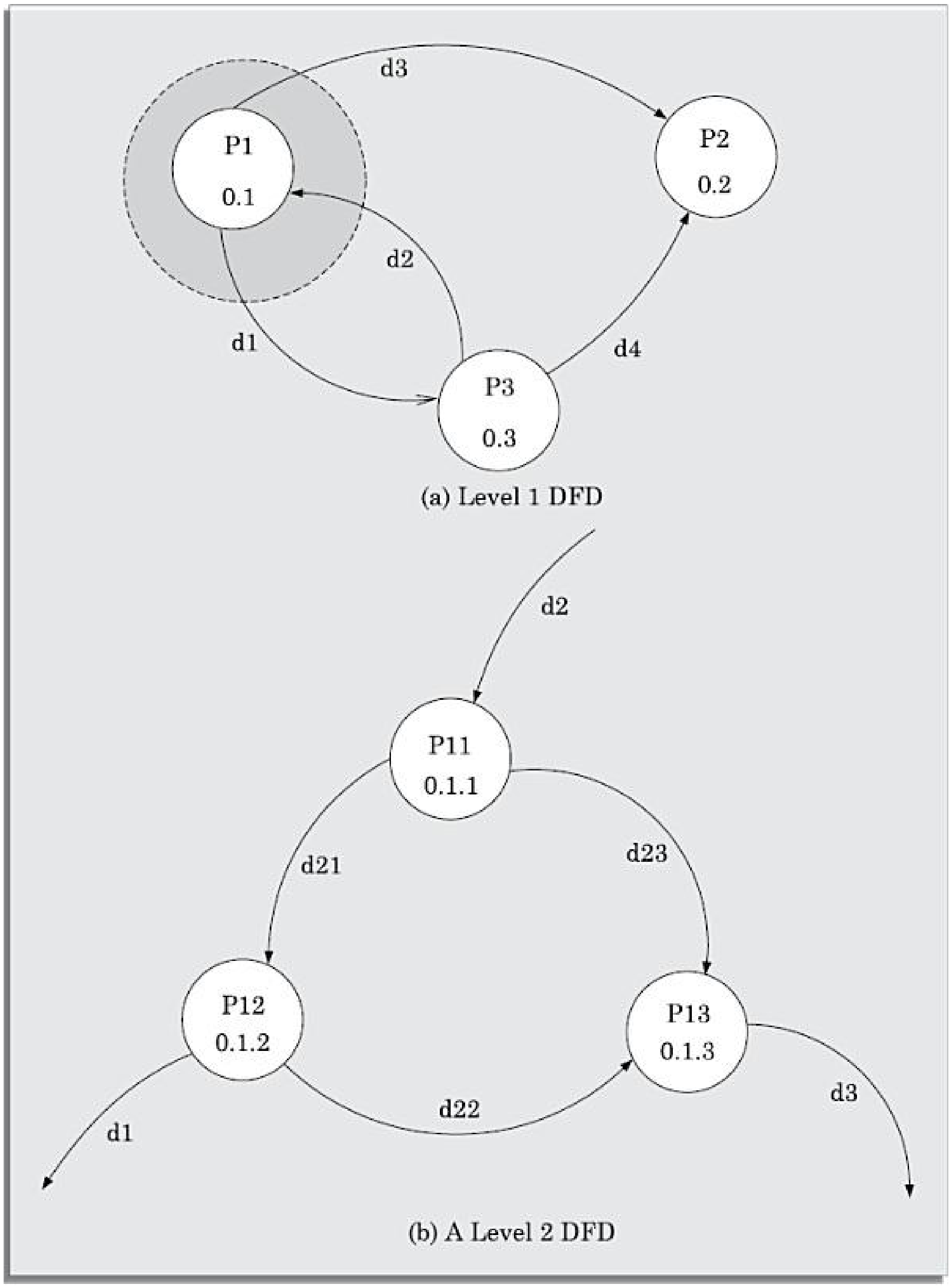


Figure 6.5: An example showing balanced decomposition.

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.

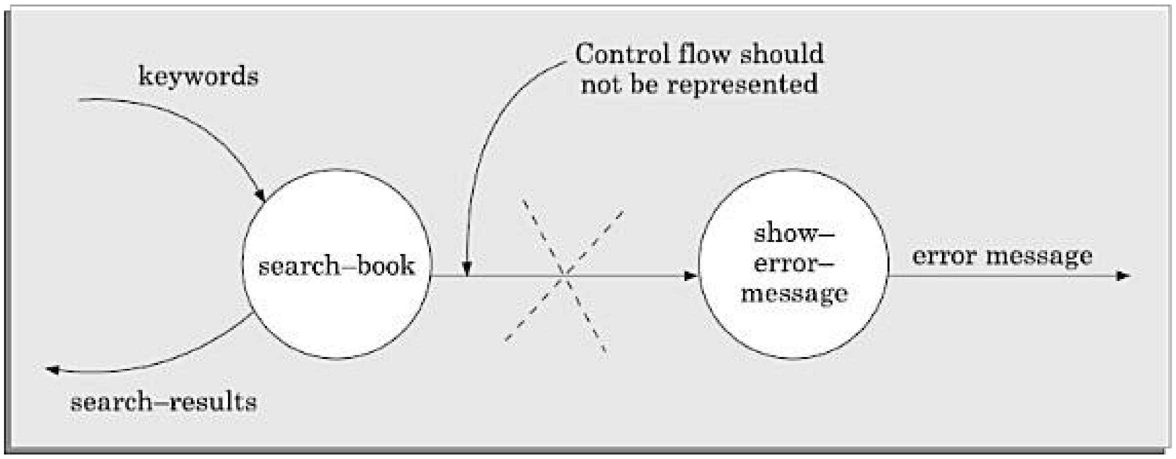


Figure 6.6: It is incorrect to show control information on a DFD.

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.

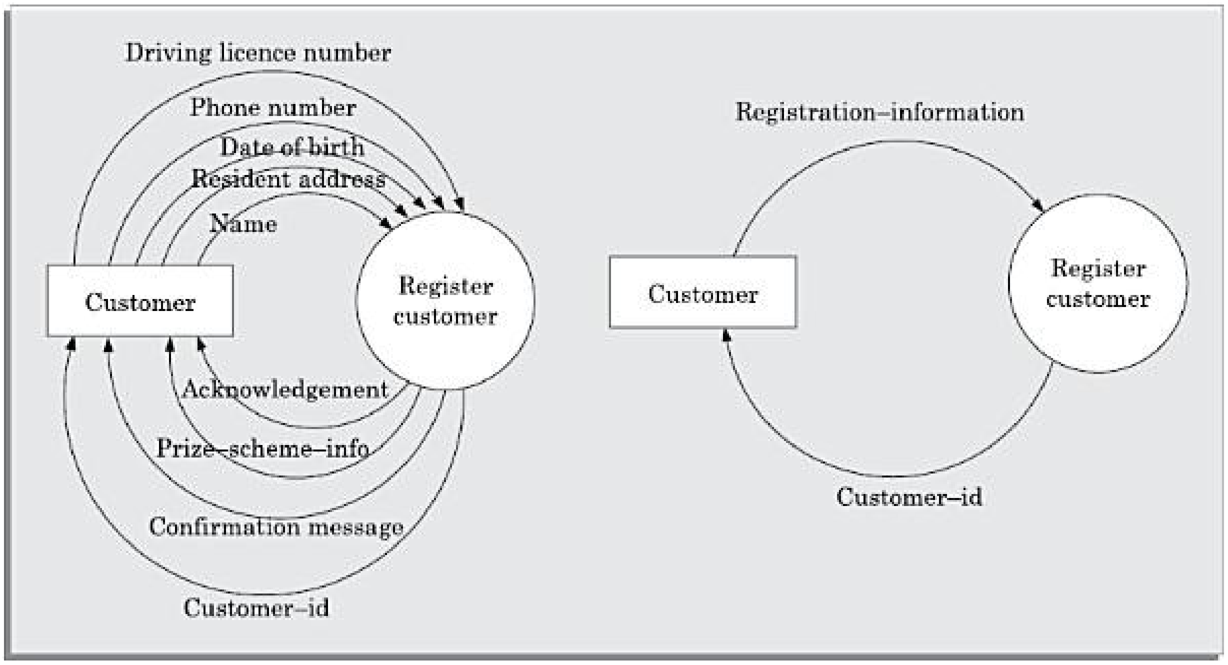


Figure 6.7: Illustration of how to avoid data cluttering.

We now illustrate the structured analysis technique through a few examples.

Example 6.1 (RMS Calculating Software) A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.

In this example, the context diagram is simple to draw. The system accepts three integers from the user and returns the result to him. This has been shown in Figure 6.8(a). To draw the level 1 DFD, from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform—accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result. After representing these four functions in Figure 6.8(b), we observe that the calculation of root mean square essentially consists of the functions—calculate the squares of the input numbers, calculate the mean, and finally calculate the root. This decomposition is shown in the level 2 DFD in Figure 6.8(c).

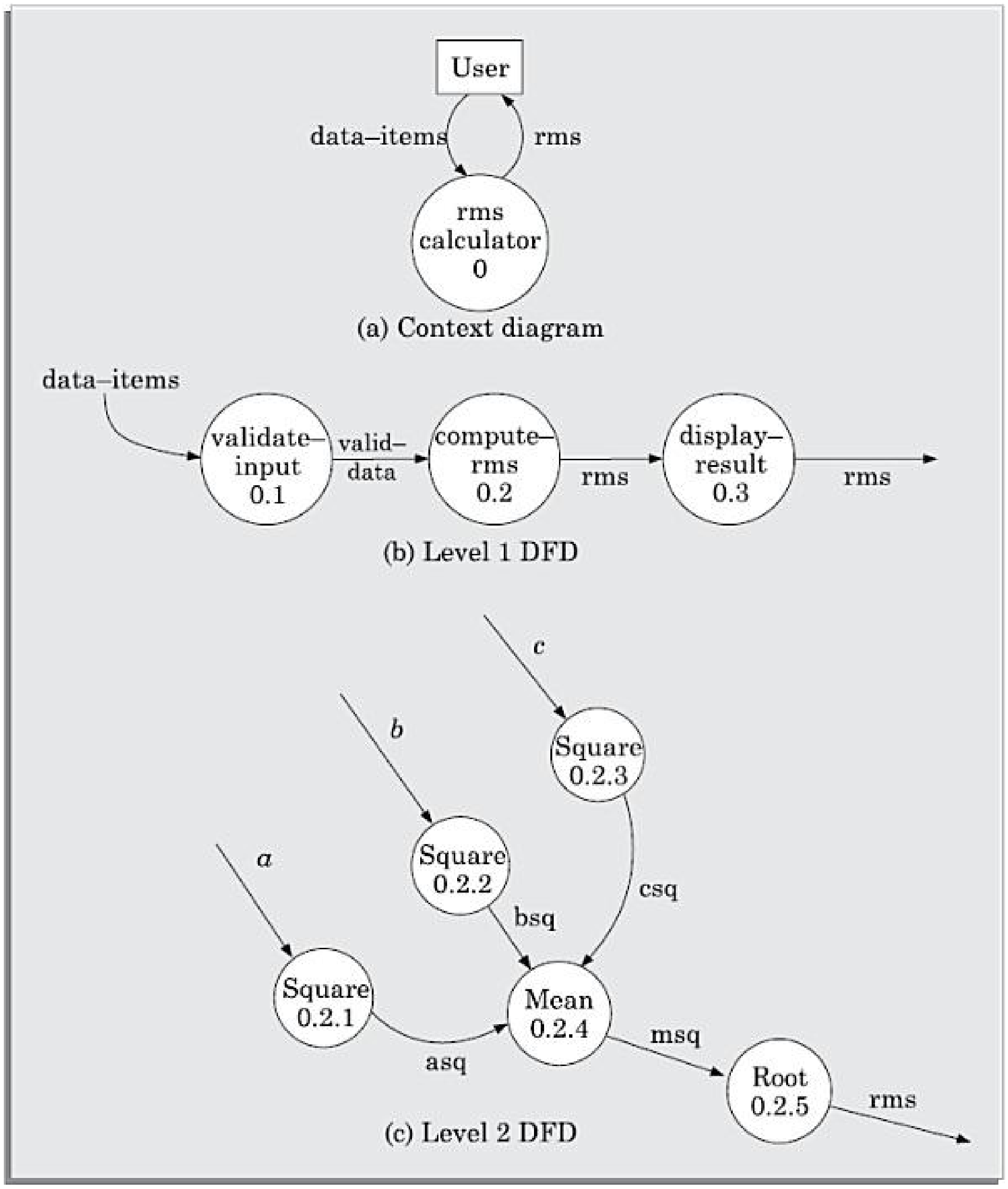


Figure 6.8: Context diagram, level 1, and level 2 DFDs for Example 6.1.

Data dictionary for the DFD model of Example 6.1

data-items: {integer}3
rms: float
valid-data:data-items
a: integer
b: integer
c: integer
asq: integer

bsq: integer
csq: integer
msq: integer

Example 6.1 is an almost trivial example and is only meant to illustrate the basic methodology. Now, let us perform the structured analysis for a more complex problem.

Example 6.2 (Tic-Tac-Toe Computer Game) Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. A move consists of marking a previously unmarked square. The player who is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. As soon as either of the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The context diagram and the level 1 DFD are shown in Figure 6.9.

Data dictionary for the DFD model of Example 6.2

move: integer /* number between 1 to 9 */
display: game+result
game: board
board: {integer}9
result: ["computer won", "human won", "drawn"]

Example 6.3 (Supermarket Prize Scheme) A super market needs to develop a software that would help it to automate a scheme that it plans to introduce to encourage regular customers. In this scheme, a customer would have first register by supplying his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN 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.

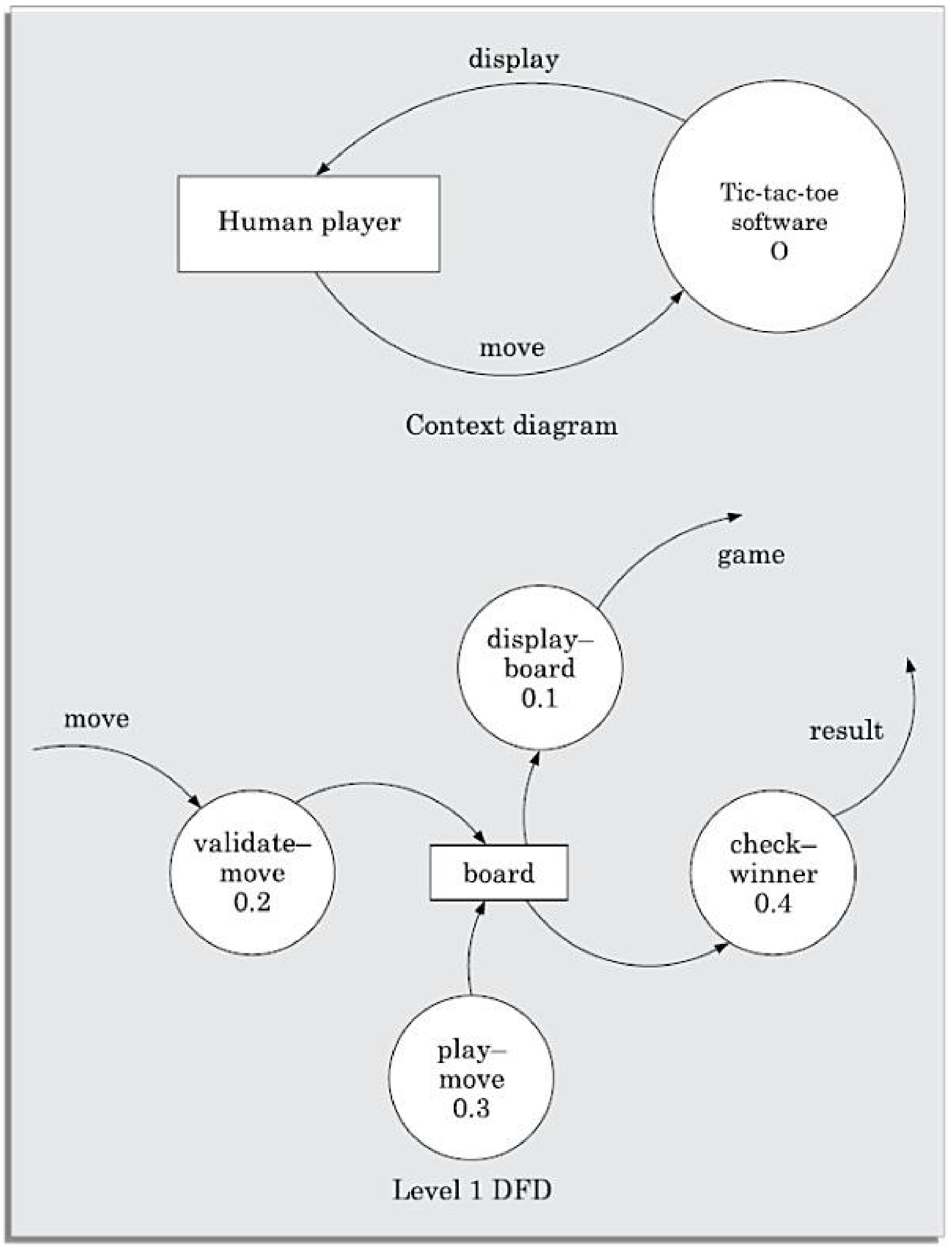


Figure 6.9: Context diagram and level 1 DFDs for Example 6.2.

The context diagram for the supermarket prize scheme problem of Example 6.3 is shown in Figure 6.10. The level 1 DFD in Figure 6.11. The level 2 DFD in Figure 6.12.

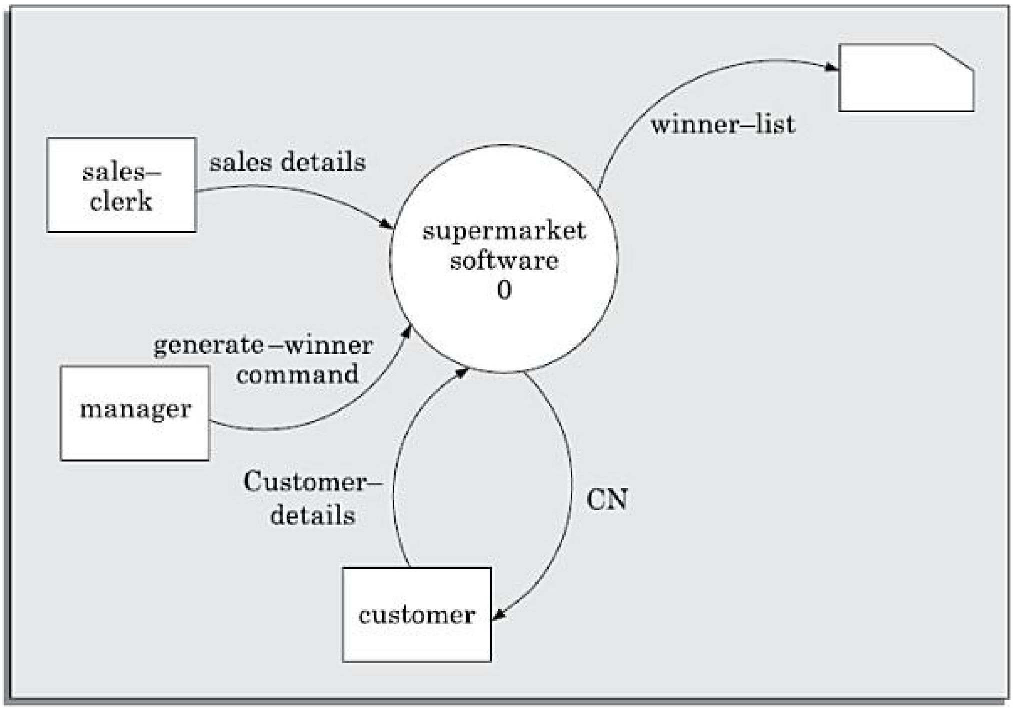


Figure 6.10: Context diagram for Example 6.3.

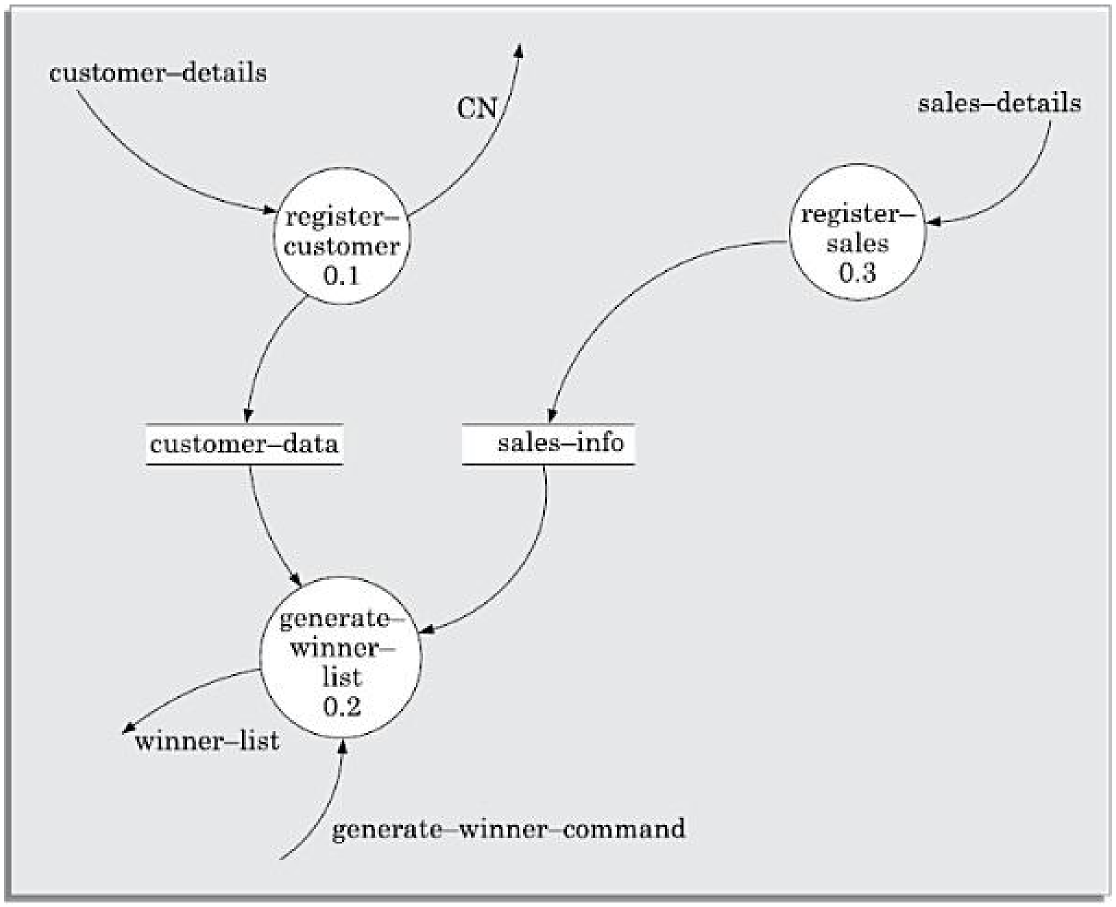


Figure 6.11: Level 1 diagram for Example 6.3.

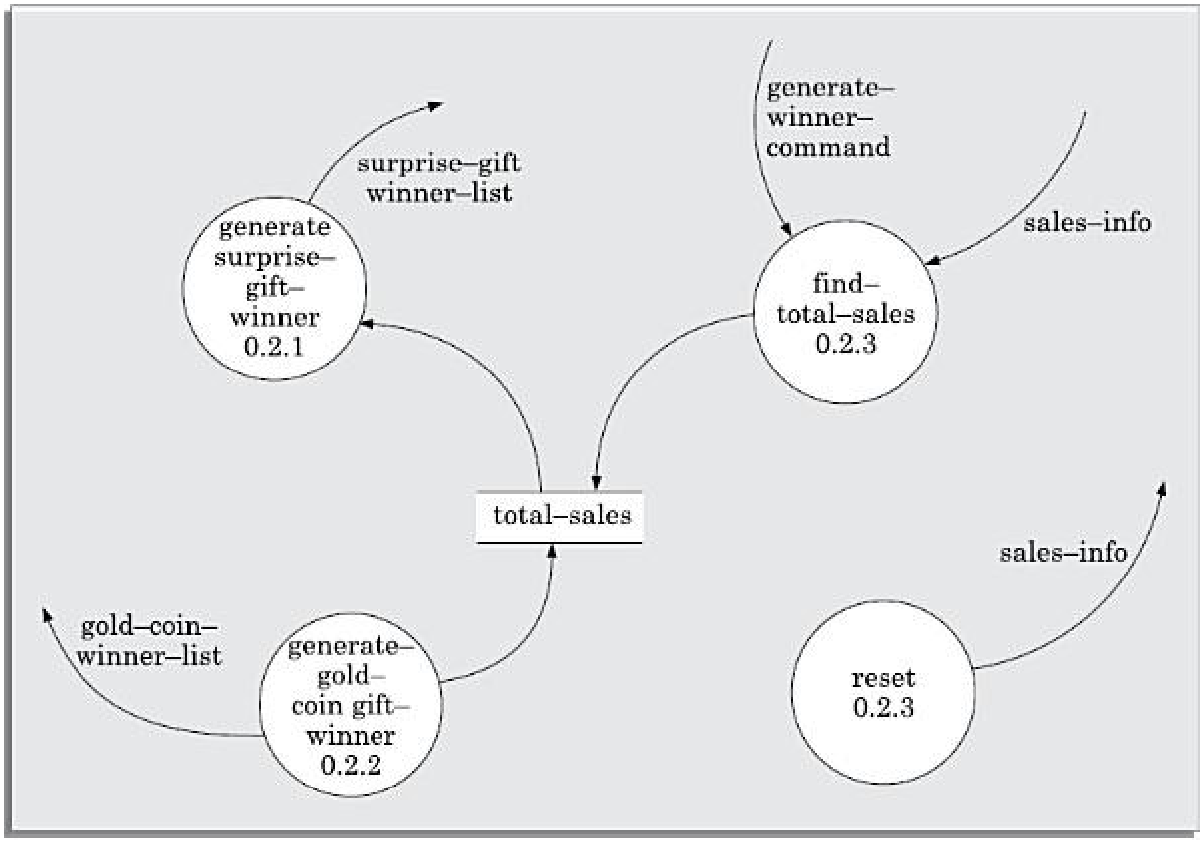


Figure 6.12: Level 2 diagram for Example 6.3.

Data dictionary for the DFD model of Example 6.3

address: name+house#+street#+city+pin
sales-details: {item+amount}* + CN
CN: integer
customer-data: {address+CN}*
sales-info: {sales-details}*
winner-list: surprise-gift-winner-list + gold-coin-winner-list
surprise-gift-winner-list: {address+CN}*
gold-coin-winner-list: {address+CN}*
gen-winner-command: command
total-sales: {CN+integer}*

Observations: The following observations can be made from the Example 6.3.

1. The fact that the customer is issued a manually prepared customer identity card or that the customer hands over the identity card each time he makes a purchase has not been shown in the DFD. This is because these are item transfers occurring outside the computer.
2. The data generate-winner-list in a way represents control information

(that is, command to the software) and no real data. We have included it in the DFD because it simplifies the structured design process as we shall realize after we practise solving a few problems. We could have also as well done without the generate-winner-list data, but this could have a bit complicated the design.

3. Observe in Figure 6.11 that we have two separate stores for the customer data and sales data. Should we have combined them into a single data store? The answer is—No, we should not. If we had combined them into a single data store, the structured design that would be carried out based on this model would become complicated. Customer data and sales data have very different characteristics. For example, customer data once created, does not change. On the other hand, the sales data changes frequently and also the sales data is reset at the end of a year, whereas the customer data is not.

Example 6.4 (Trading-house Automation System (TAS)) A trading house wants us to develop a computerized system that would automate various book-keeping activities associated with its business. The following are the salient features of the system to be developed:

- The trading house has a set of regular customers. The customers place orders with it for various kinds of commodities. The trading house maintains the names and addresses of its regular customers. Each of these regular customers should be assigned a unique customer identification number (CIN) by the computer. The customers quote their CIN on every order they place.
- Once order is placed, as per current practice, the accounts department of the trading house first checks the credit-worthiness of the customer. The credit-worthiness of the customer is determined by analysing the history of his payments to different bills sent to him in the past. After automation, this task has be done by the computer.
- If a customer is not credit-worthy, his orders are not processed any further and an appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy, the items that he has ordered are checked against the list of items that the trading house deals with. The items in the order which the trading house does not deal with, are not processed any further and an appropriate apology message for the

customer for these items is generated.

- The items in the customer's order that the trading house deals with are checked for availability in the inventory. If the items are available in the inventory in desired quantity, then:

– A bill is with the forwarding address of the customer is printed.
– A material issue slip is printed. The customer can produce this material issue slip at the store house and take delivery of the items.
– Inventory data is adjusted to reflect the sale to the customer.

- If any of the ordered items are not available in the inventory in sufficient quantity to satisfy the order, then these out-of-stock items along with the quantity ordered by the customer and the CIN are stored in a "pending-order" file for further processing to be carried out when the purchase department issues the "generate indent" command.
- The purchase department should be allowed to periodically issue commands to generate indents. When a command to generate indents is issued, the system should examine the "pending-order" file to determine the orders that are pending and determine the total quantity required for each of the items. It should find out the addresses of the vendors who supply these items by examining a file containing vendor details and then should print out indents to these vendors.
- The system should also answer managerial queries regarding the statistics of different items sold over any given period of time and the corresponding quantity sold and the price realised.

The context diagram for the trading house automation problem is shown in Figure 6.13. The level 1 DFD in Figure 6.14.

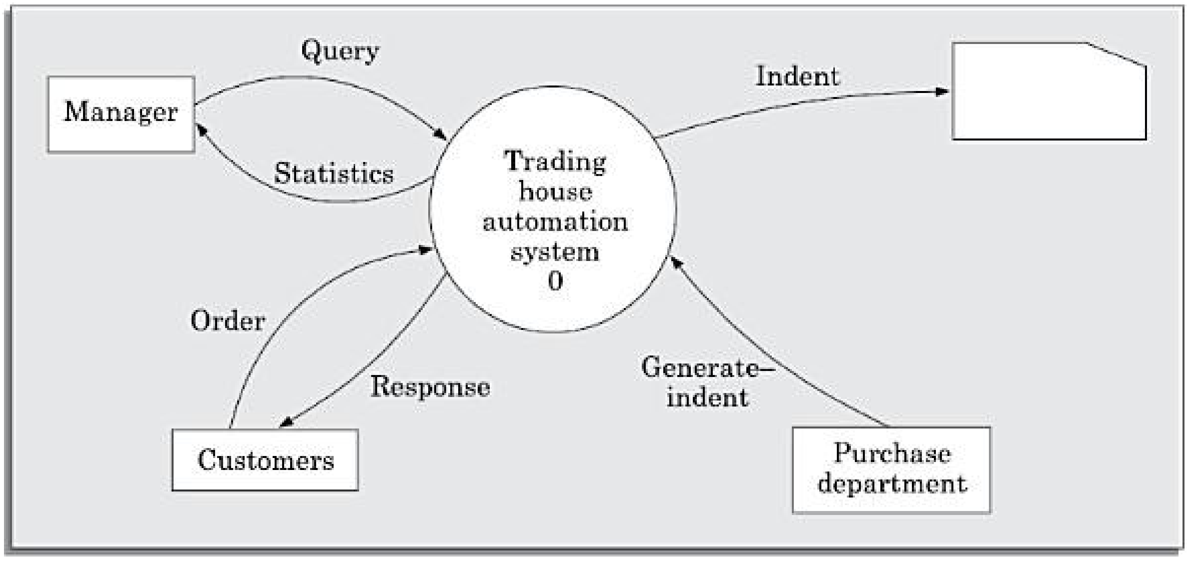


Figure 6.13: Context diagram for Example 6.4.

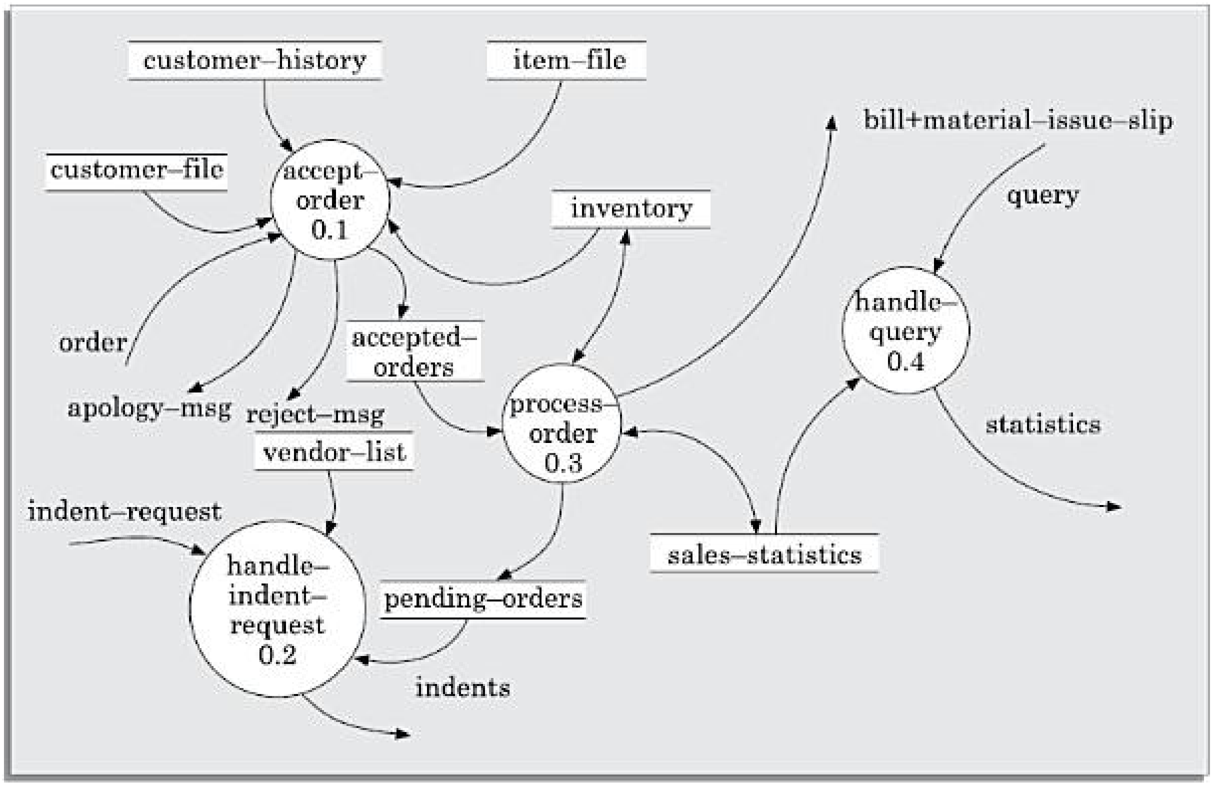


Figure 6.14: Level 1 DFD for Example 6.4.

Data dictionary for the DFD model of Example 6.4

response: [bill + material-issue-slip, reject-msg,apology-msg]
query: period /* query from manager regarding sales statistics*/
period: [date+date,month,year,day]
date: year + month + day year: integer
month: integer day: integer customer-id: integer
order: customer-id + {items + quantity}* + order#

accepted-order: order /* ordered items available in inventory */
reject-msg: order + message /* rejection message */
pending-orders: customer-id + order# + {items+quantity}*
customer-address: name+house#+street#+city+pin
name: string
house#: string
street#: string
city: string
pin: integer
customer-id: integer
customer-file: {customer-address}* + customer-id
bill: {item + quantity + price}* + total-amount + customer-address + order#
material-issue-slip: message + item + quantity + customer-address
message: string
statistics: {item + quantity + price }*
sales-statistics: {statistics}* + date
quantity: integer
order#: integer /* unique order number generated by the program */
price: integer
total-amount: integer
generate-indent: command
indent: {item+quantity}* + vendor-address
indents: {indent}*
vendor-address: customer-address
vendor-list: {vendor-address}*
item-file: {item}*
item: string
indent-request: command

Observations: The following observations can be made from Example 6.4.

1. In a DFD, if two data stores deal with different types of data, e.g. one type of data is invariant with time whereas another varies with time, (e.g. vendor address, and inventory data) it is a good idea to represent them as separate data stores.

If two types of data always get updated at the same time, they should be stored in a single data store. Otherwise, separate data stores should be used for them.

The inventory data changes each time supply arrives and the inventory is updated or an item is sold, whereas the vendor data remains unchanged.

2. If we are developing the DFD model of a process which is already being manually carried out, then the names of the registers being maintained in the manual process would appear as data stores in the DFD model. For example, if TAS is currently being manually carried out, then normally there would registers corresponding to accepted orders, pending orders, vendor list, etc.
3. We can observe that DFDs enable a software developer to develop the data domain and functional domain model of the system at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition. At the same time, the DFD refinement automatically results in refinement of corresponding data items.
4. The data that are maintained in physical registers in manual processing, become data stores in the DFD representation. Therefore, to determine which data should be represented as a data store, it is useful to try to imagine whether a set of data items would be maintained in a register in a manual system.

Example 6.5 (Personal Library Software) Perform structured analysis for the personal library software of Example 6.5.

The context diagram is shown in Figure 6.15.

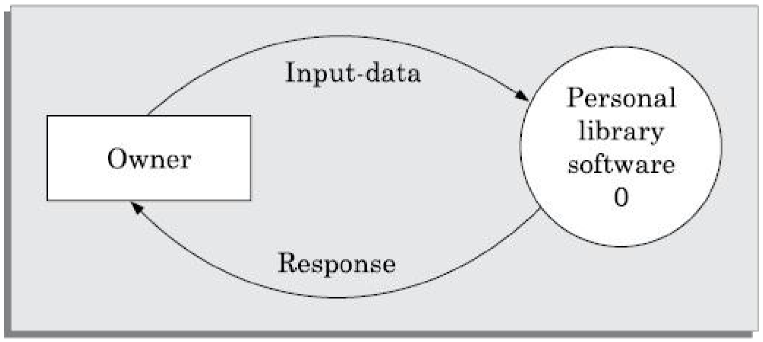


Figure 6.15: Context diagram for Example 6.5.

The level 1 DFD is shown in Figure 6.16.

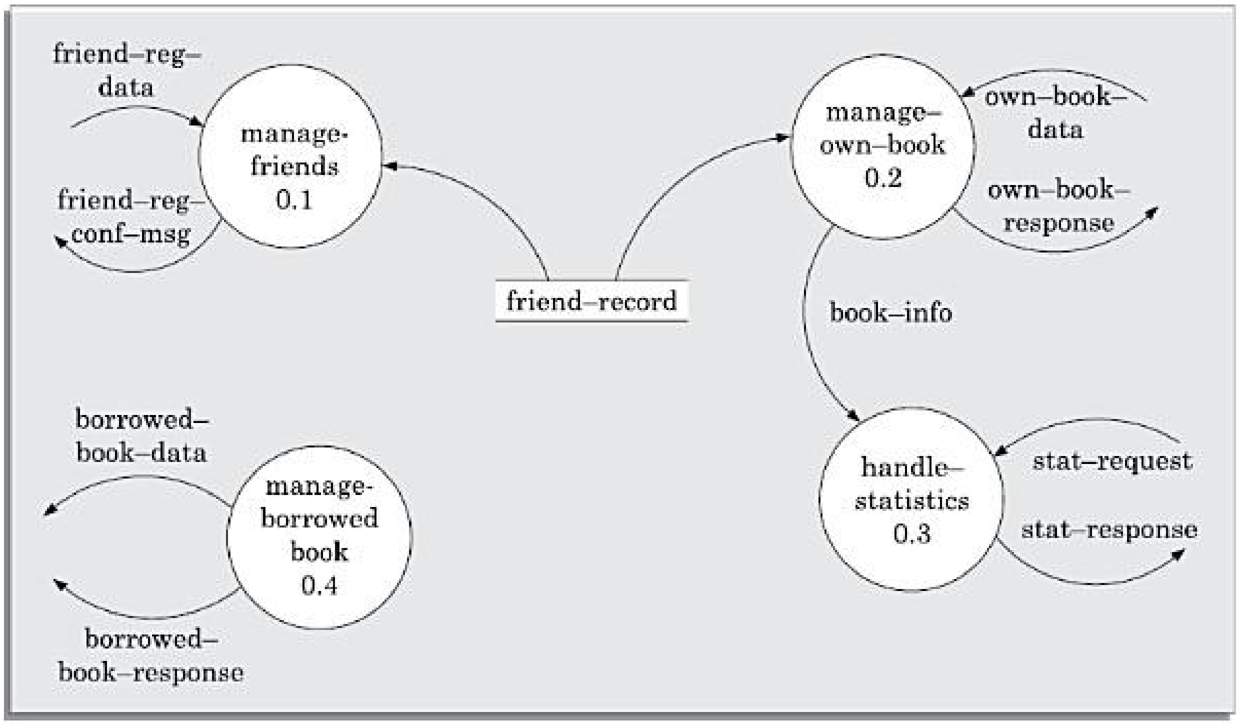


Figure 6.16: Level 1 DFD for Example 6.5.

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

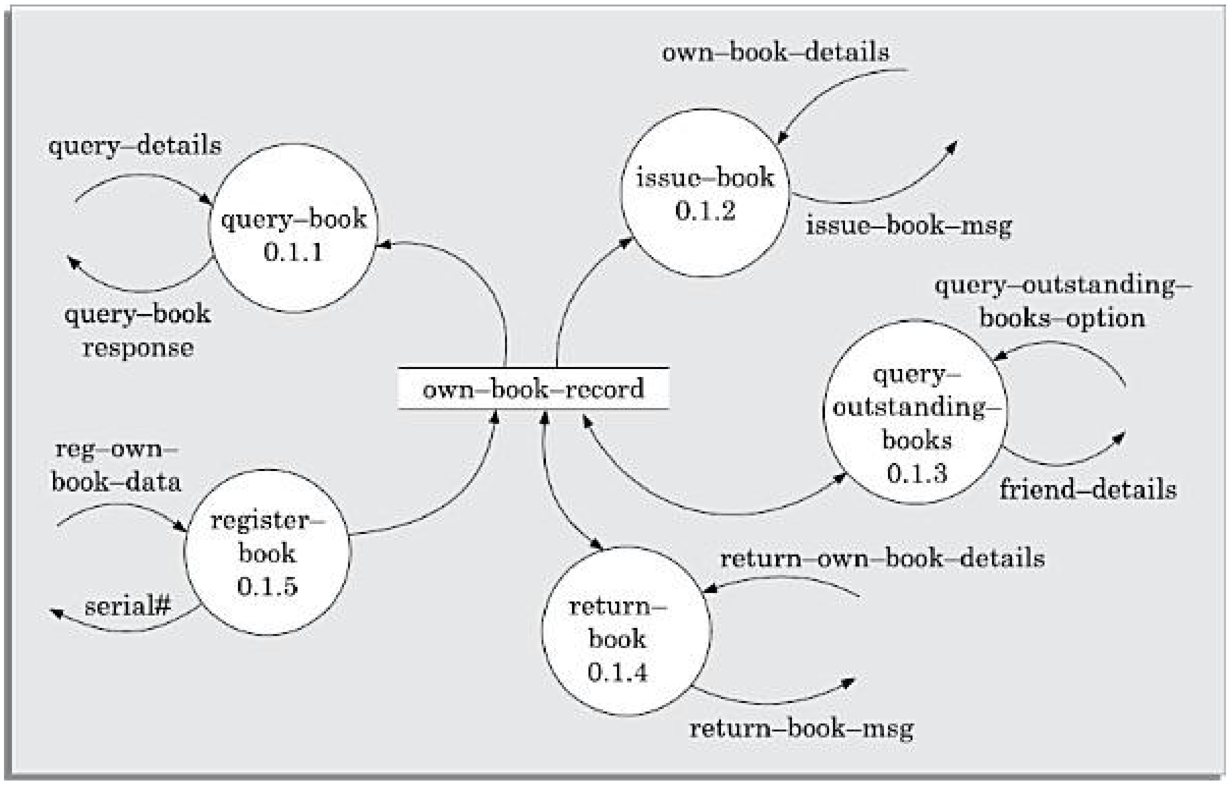


Figure 6.17: Level 2 DFD for Example 6.5.

Data dictionary for the DFD model of Example 6.5

input-data: friend-reg-data + own-book-data + stat-request + borrowed-book-data

response: friend-reg-conf-msg + own-book-response + stat-response + borrowed-book-response
own-book-data: query-details + own-book-details + query-outstanding-books-option + return-own book-details + reg-own-book-data
own-book-response: query-book-response + issue-book-msg + friend-details + return-book- msg + serial#.
borrowed-book-data: borrowed-book-details + book-return-details + display-books-option borrowed-book-response: reg-msg + unreg-msg + borrowed-books-list
friend-reg-data: name + address + landline# + mobile#
own-book-details: friend-reg-data + book-title + data-of-issue
return-own-book-details: book-title + date-of-return
friend-details: name + address + landline# + mobile# + book-list
borrowed-book-details: book-title + borrow-date
serial#: integer

Observation: Observe that since there are more than seven functional requirements for the personal library software, related requirements have been combined to have only five bubbles in the level 1 diagram. Only level 2 DFD has been shown, since the other DFDs are trivial and need not be drawn.

Shortcomings of the DFD model

DFD models suffer from several shortcomings. The important shortcomings of DFD models are the following:

- Imprecise DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information i s missing or is incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Not-well defined control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modelling real-time systems.
- Decomposition: The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is

superior or preferable to another one.

- Improper data flow diagram: The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition.

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 two 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.

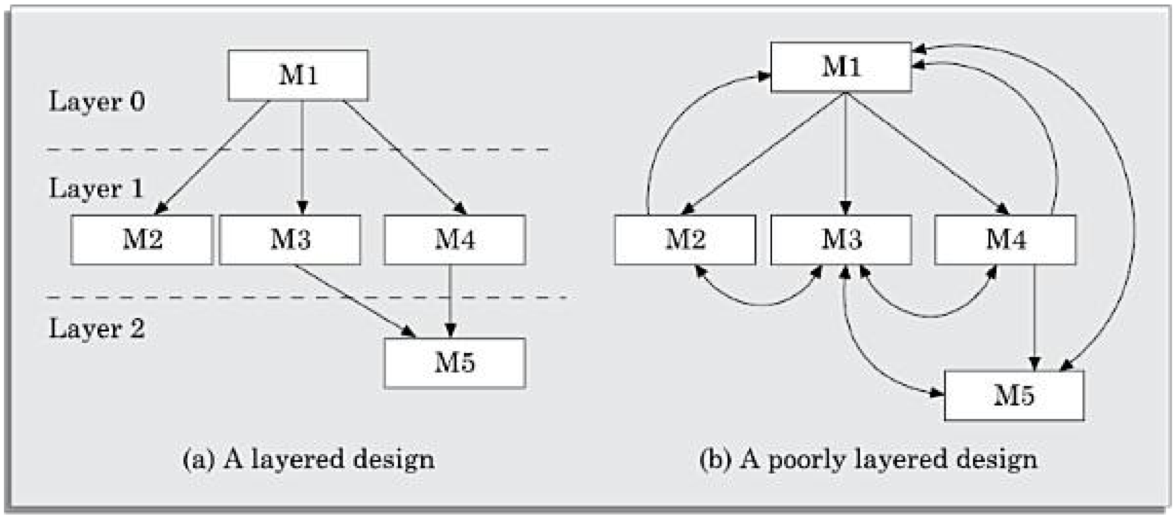


Figure 6.18: Examples of properly and poorly layered designs.

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 flow 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 is 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

Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs.

At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

Given a specific DFD of a model, how does one decide whether to apply transform analysis or transaction analysis? For this, one would have to examine the data input to the diagram. The data input to the diagram can be easily spotted because they are represented by dangling arrows. If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing.

Please recollect that the bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code. Therefore, transform analysis is normally applicable at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

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. The first level of 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.

Example 6.6 Draw the structure chart for the RMS software of Example 6.1.

By observing the level 1 DFD of Figure 6.8, we can identify validate-input as the afferent branch and write-output as the efferent branch. The remaining (i.e., compute-rms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in Figure 6.19.

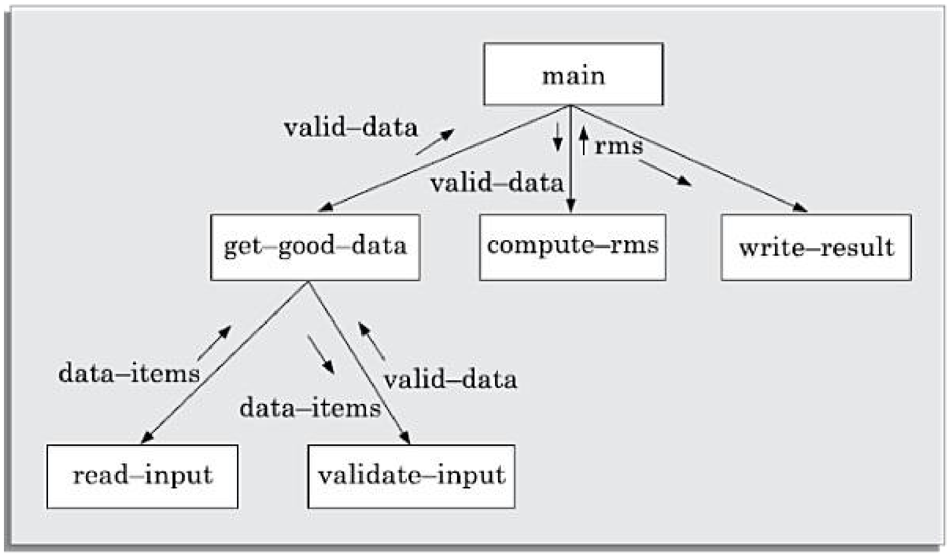


Figure 6.19: Structure chart for Example 6.6.

Example 6.7 Draw the structure chart for the tic-tac-toe software of Example 6.2.

The structure chart for the Tic-tac-toe software is shown in Figure 6.20. Observe that the check-game-status bubble, though produces some outputs. is not really responsible for converting logical data to physical data. On the other hand, it carries out the processing involving checking game status. That is the main reason, why we have considered it as a central transform and not as an efferent type of module.

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.

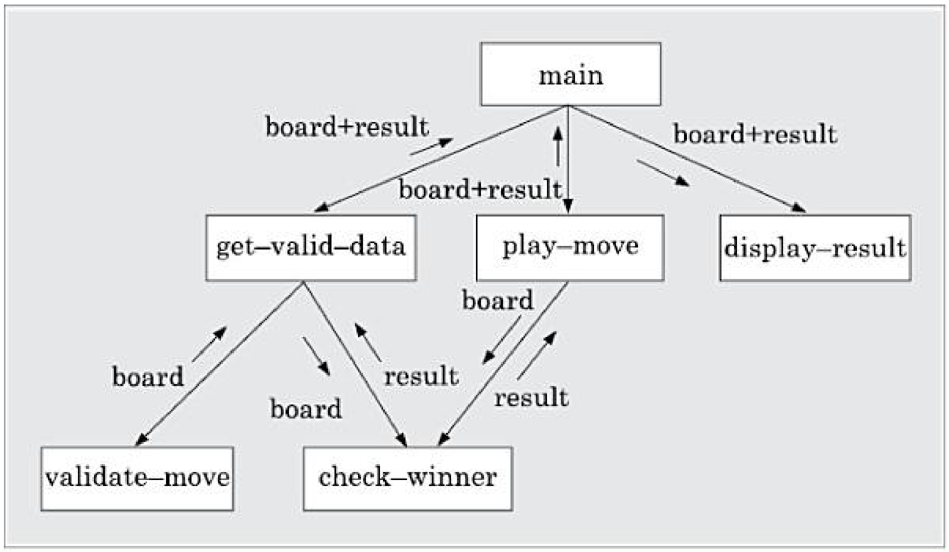


Figure 6.20: Structure chart for Example 6.7.

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 and a transaction-center module.

Example 6.8 Draw the structure chart for the Supermarket Prize Scheme software of Example 6.3.

The structure chart for the Supermarket Prize Scheme software is shown in Figure 6.21.

Example 6.9 Draw the structure chart for the *trade-house automation system* (TAS) software of Example 6.4.

The structure chart for the *trade-house automation system* (TAS) software of

Example 6.4 is shown in Figure 6.22.

By observing the level 1 DFD of Figure 6.14, we can see that the data input to the diagram are handled by different bubbles and therefore transaction analysis is applicable to this DFD. Input data to this DFD are handled in three different ways (accept-order, accept- indent-request, and handle-query), we have three different transactions corresponding to these as shown in Figure 6.22.

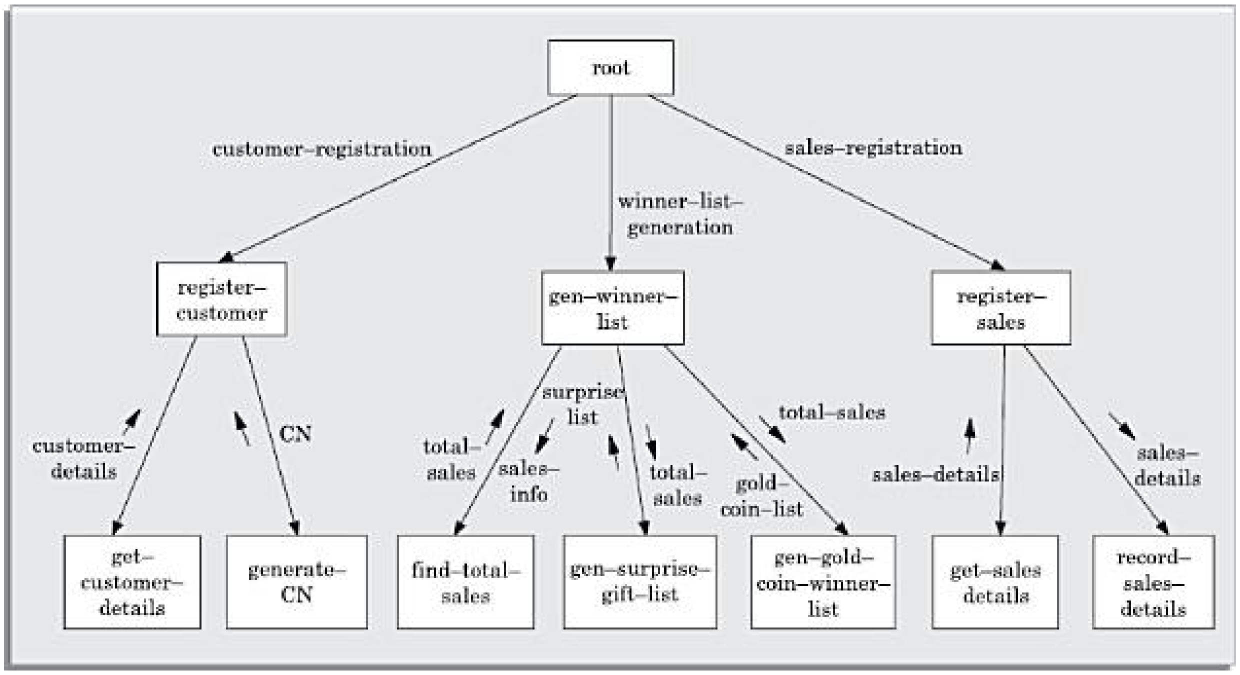


Figure 6.21: Structure chart for Example 6.8.

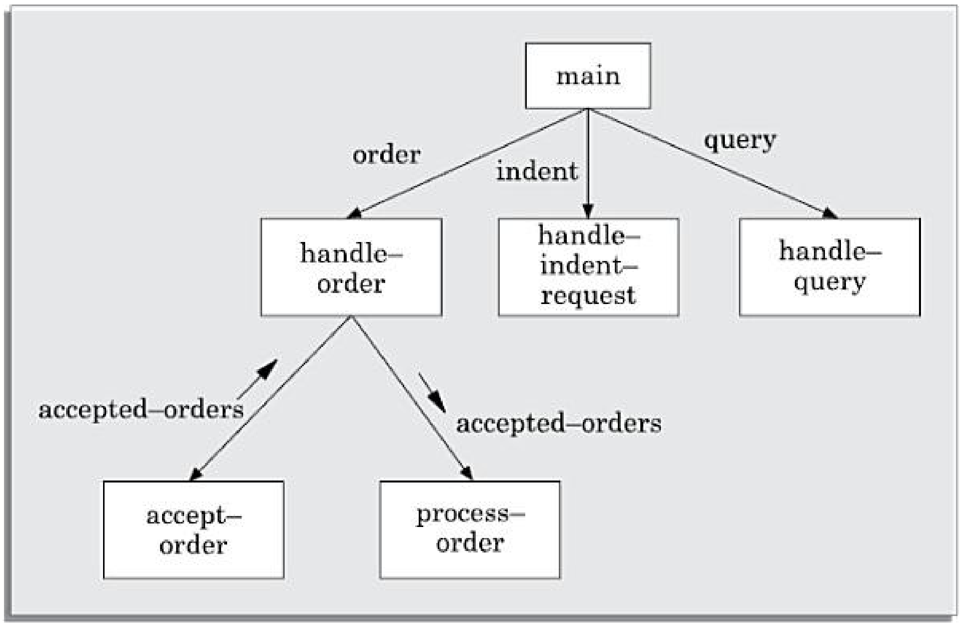


Figure 6.22: Structure chart for Example 6.9.

Word of caution

We should view transform and transaction analyses as guidelines, rather than rules. We should apply these guidelines in the context of the problem and handle the pathogenic cases carefully.

Example 6.10 Draw the structure chart for the personal library software of Example 6.6.

The structure chart for the personal library software is shown in Figure 6.23.

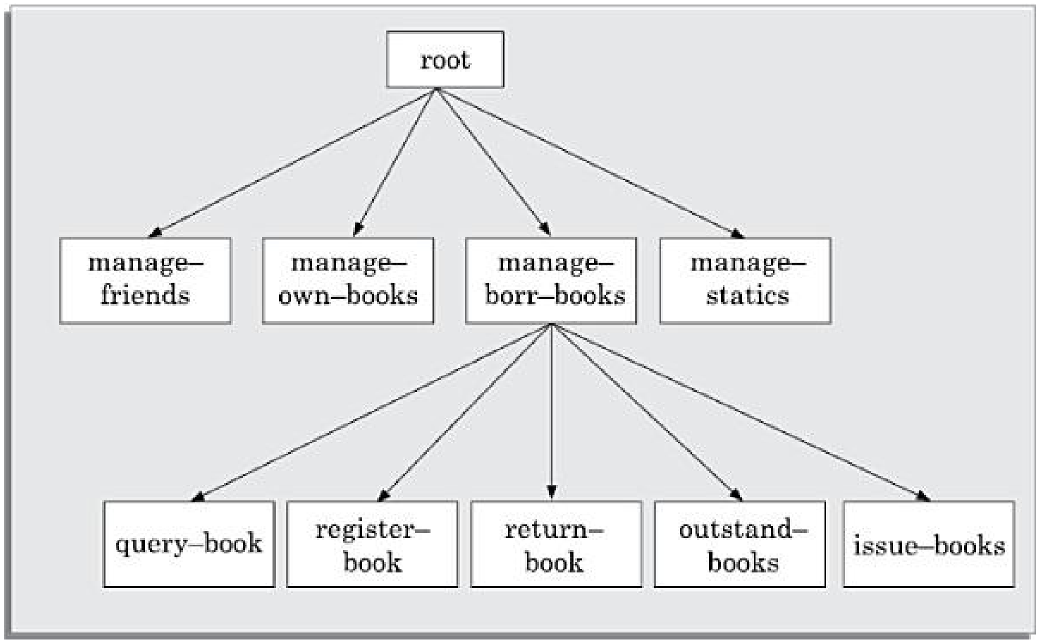


Figure 6.23: Structure chart for Example 6.10.

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 and 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.

SUMMARY

- In this chapter, we discussed a sample function-oriented software design methodology called structured analysis/structured design (SA/SD) which incorporates features of some important design methodologies.
- Methodologies like SA/SD give us a recipe for developing a good design according to the different goodness criteria we had discussed in Chapter 5. item SA/SD consists of two important parts—structured analysis and structured design.
- The goal of structured analysis is to perform a functional decomposition of the system. Results of structured analysis is represented using data flow diagrams (DFDs). The DFD representation is difficult to implement using a traditional programming language. The DFD representation can be systematically be transformed to structure chart representation. The structure chart representation can be easily implemented using a conventional programming language.
- During structured design, the DFD representation obtained during

Chapter

9

USER INTERFACE DESIGN

The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface (can you think of a software product that does not have any user interface?). In the early days of computer, no software product had any user interface. The computers those days were batch systems and no interactions with the users were supported. Now, we know that things are very different—almost every software product is highly interactive. The user interface part of a software product is responsible for all interactions with the end-user. Consequently, the user interface part of any software product is of direct concern to the end-users. No wonder then that many users often judge a software product based on its user interface. Aesthetics apart, an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction. Users become particularly irritated when a system behaves in an unexpected ways, i.e., issued commands do not carry out actions according to the intuitive expectations of the user. Normally, when a user starts using a system, he builds a mental model of the system and expects the system behaviour to conform to it. For example, if a user action causes one type of system activity and response under some context, then the user would expect similar system activity and response to occur for similar user actions in similar contexts. Therefore, sufficient care and attention should be paid to the design of the user interface of any software product.

Systematic development of the user interface is also important from another consideration. Development of a good user interface usually takes significant portion of the total system development effort. For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part. Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously. Therefore, it is necessary to carefully study various concepts associated with user interface design and understand various systematic techniques available for the development of user interface.

In this chapter, we first discuss some common terminologies and concepts associated with development of user interfaces. Then, we classify the different types of interfaces commonly being used. We also provide some guidelines for designing good interfaces, and discuss some tools for development of graphical user interfaces (GUIs). Finally, we present a GUI development methodology.

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:

— **Use 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. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

— **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. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of a menu to minimise the mouse movements necessary to issue commands.

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. When someone uses an application for the first time, his primary concern is speed of learning. After using an application for extended periods of time, he becomes familiar with the operation of the software. As a user becomes more and more familiar with an interface, his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot-keys”, “macros”, etc.

Thus, the skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

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 BASIC CONCEPTS

In this section, we first discuss some basic concepts in user guidance and on-line help system. Next, we examine the concept of a mode-based and a modeless interface and the advantages of a graphical interface.

9.2.1 User Guidance and On-line Help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.

On-line help system: Users expect the on-line help messages to be tailored to the context in which they invoke the "help system". Therefore, a good on-line help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user's experience level. Further, a good on-line help system should take advantage of any graphics and animation characteristics of the screen and should not just be a copy of the user's manual.

Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface (These different types of interfaces are discussed later in this chapter). Also, users should have an option to turn off the detailed messages.

Error messages: Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc. Users do not like error messages that are either ambiguous or too general such as "invalid input or system error". Error messages should be polite. Error messages should not have associated noise which might embarrass the user. The message should suggest how a given error can be rectified. If appropriate, the user should be given the option of invoking the on-line help system to find out more about the error situation.

9.2.2 Mode-based versus Modeless Interface

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is, i.e., the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

9.2.3 Graphical User Interface (GUI) versus Text-based User Interface

Let us compare various characteristics of a GUI with those of a text-based user interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of the biggest advantages of GUI over text- based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows.

- Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash for deleting is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy of command issue procedure.
- On the flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse. On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals. However, display terminals with graphics capability with bit-mapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops. Therefore, the emphasis of this chapter is on GUI design rather than text-based user interface design.

9.3 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.

9.3.1 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.

9.3.2 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.

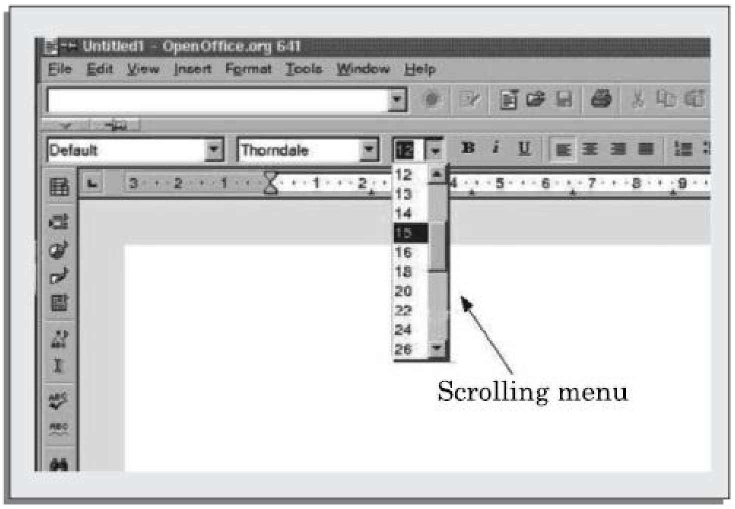

Untitled1 - OpenOffice.org 641
File Edit View Insert Format Tools Window Help
Default
Thorndale
12
12
13
14
15
16
18
20
22
24
26
Scrolling menu

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.

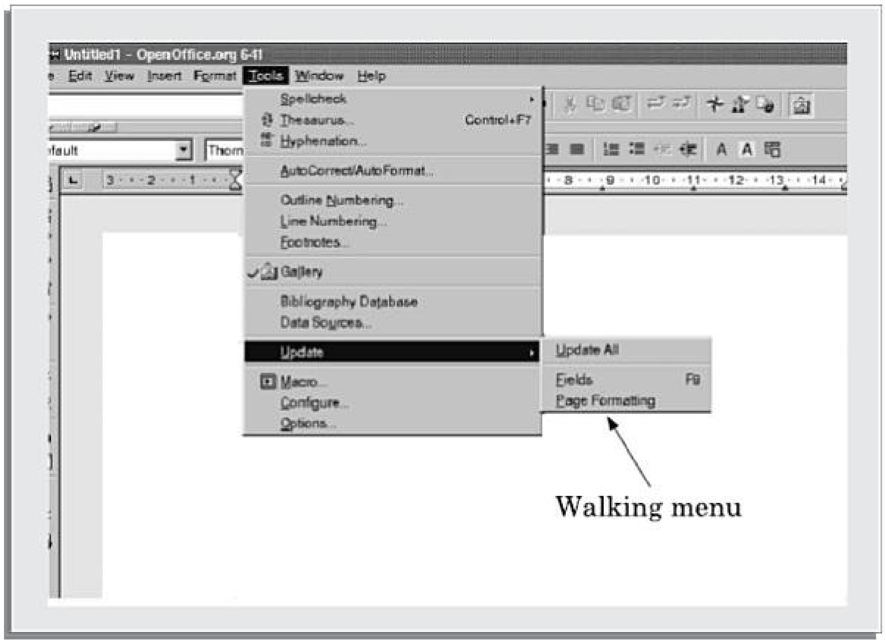


Figure 9.2: Example of walking menu.

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.

9.3.3 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 *.*.

9.4 FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

Graphical user interfaces became popular in the 1980s. The main reason why there were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive. For example, the price of a graphics terminal those days was much more than what a high-end personal computer costs these days. Also, the graphics terminals were of storage tube type and lacked raster capability.

One of the first computers to support GUI-based applications was the Apple Macintosh computer. In fact, the popularity of the Apple Macintosh computer in the early eighties is directly attributable to its GUI. In those early days of GUI design, the user interface programmer typically started his interface development from the scratch. He would starting from simple pixel display routines, write programs to draw lines, circles, text, etc. He would then develop his own routines to display menu items, make menu choices, etc. The current user interface style has undergone a sea change compared to the early style.

The current style of user interface development is component-based. It recognises that every user interface can easily be built from a handfuls of predefined components such as menus, dialog boxes, forms, etc. Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

In the following subsections, we provide an overview of the window management system, the component-based development style, and visual programming.

9.4.1 Window System

Most modern graphical user interfaces are developed using some window system. A window system can generate displays through a set of windows. Since a window is the basic entity in such a graphical user interface, we need to first discuss what exactly a window is.

Window: A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.

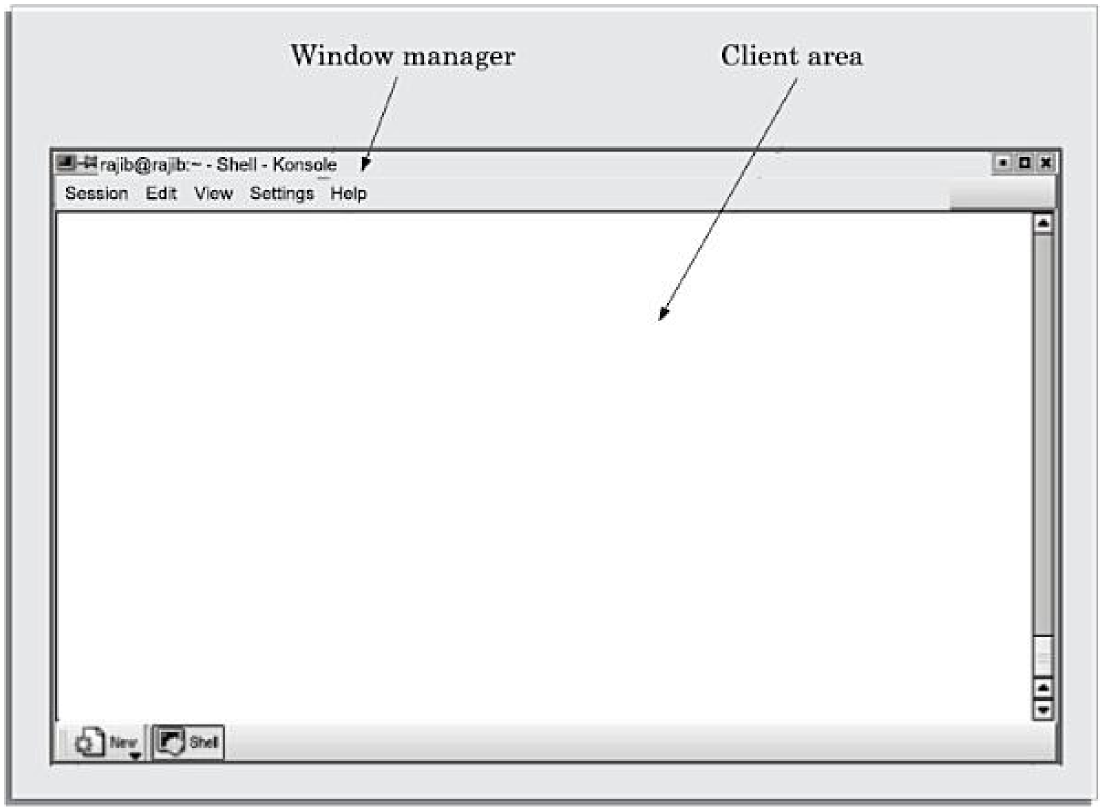

Window manager
Client area
rajib@rajib:~ - Shell - Konsole
Session Edit View Settings Help
New
Shell

Figure 9.3: Window with client and user areas marked.

A window can be divided into two parts—client part, and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client-part of the window determines the look and feel of the window. The look and feel defines a basic behaviour for all windows, such as creating, moving, resizing, iconifying of the windows. The window manager is responsible for managing and maintaining the non-client area of a window. A basic window with its different parts is shown in Figure 9.3.

Window management system (WMS)

A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS). A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS)—which not only does resource management, but also provides the basic behaviour to the windows and provides several utility routines to the application programmer for user interface development. A WMS simplifies the task of a GUI designer to a great extent by providing the basic behaviour to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application program such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.

A WMS consists of two parts (see Figure 9.4):

- a window manager, and
- a window system.

These components of the WMS are discussed in the following subsection.

Window manager and window system: The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determines how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system. The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure 9.4. This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manger invoke services of the window manager.

Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

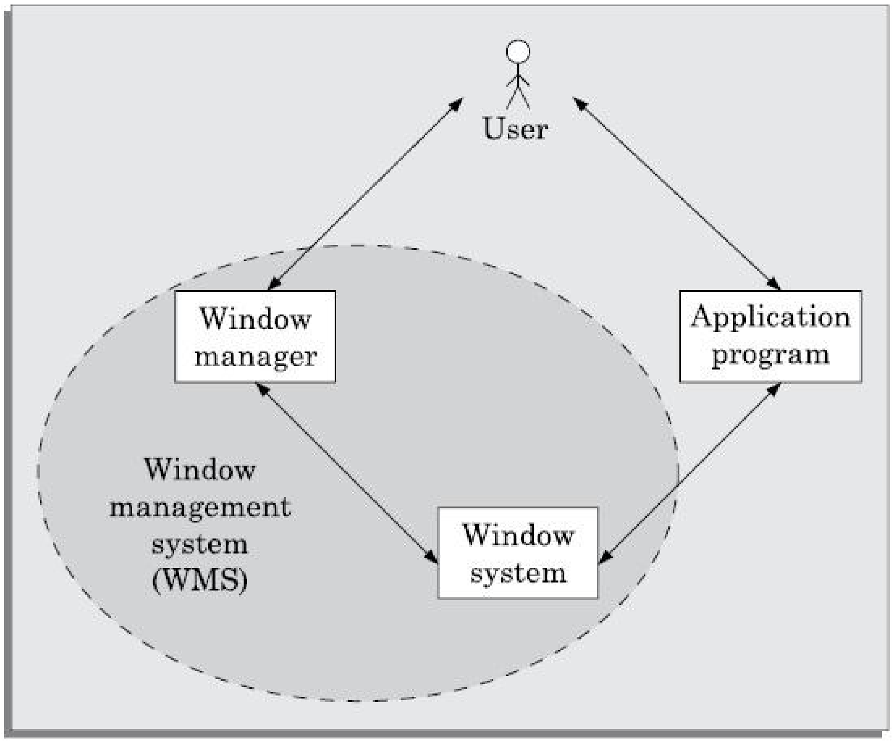


Figure 9.4: Window management system.

It is usually cumbersome to develop user interfaces using the large set of routines provided by the basic window system. Therefore, most user interface development systems usually provide a high-level abstraction called widgets for user interface development. A widget is the short form of a window object. We know that an object is essentially a collection of related data with several operations defined on these data which are available externally to operate on these data. The data of an window object are the geometric attributes (such as size, location etc.) and other attributes such as its background and foreground colour, etc. The operations that are defined on these data include, resize, move, draw, etc.

Widgets are the standard user interface components. A user interface is usually made up by integrating several widgets. A few important types of widgets normally provided with a user interface development system are described in Section 9.4.2.

Component-based development

A development style based on widgets is called component-based (or widget-based) GUI development style. There are several important advantages of using a widget-based design style. One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components. Therefore, the user can extend his knowledge of the behaviour of the standard components from one application to the other. Also, the component-based user interface development style reduces the application programmer's work significantly as he is more of a user interface component integrator than a programmer in the traditional sense. In the following section, we will discuss some of these popular widgets.

Visual programming

Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g., menu, forms, etc.) from the displayed icons and placing them wherever required. Thus, visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g., factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop the interface.

Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with window-based user interfaces for Microsoft Windows environments. In visual C++ you usually design menu bars, icons, and dialog boxes, etc. before adding them to your program. These objects are called as resources. You can design shape, location, type, and size of the dialog boxes before writing any C++ code for the application.

9.4.2 Types of Widgets

Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets, so that one can think of a generic widget set which is applicable to most interfaces. The following widgets we have chosen as representatives of this generic class.

Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

Pop-up menu: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

Pull-down menu : These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.

Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . .). A push button with an ellipsis generally indicates that another dialog box will appear.

Radio buttons: A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

9.4.3 An Overview of X-Window/MOTIF

One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs. Applications developed using the X-window system are device-independent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent GUI operation has been schematically represented in Figure 9.5. Here, A is the computer application in which the application is running. B can be any computer on the network from where you can interact with the application. Network-independent GUI was pioneered by the X-window system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Now-a-days many user interface development systems support network-independent GUI development, e.g., the AWT and Swing components of Java.

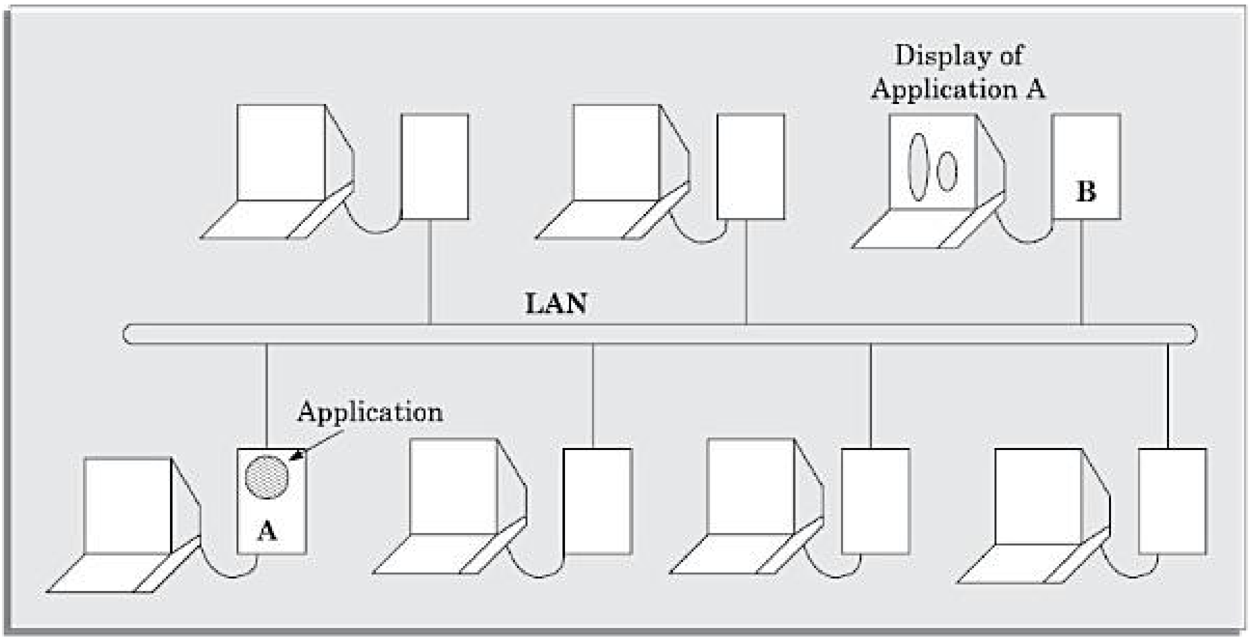


Figure 9.5: Network-independent GUI.

The X-window functions are low level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines. Built on top of X-windows are higher level functions collectively called Xtoolkit, which consists of a set of basic widgets and a set of routines to manipulate these widgets. One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface designs called DECWindows. In the following, we shall provide a very brief overview of the X-window system and its architecture and the interested reader is referred to Scheifler et al. [1988] for further study on graphical user interface development using X-windows and Motif.

9.4.4 X Architecture

The X architecture is pictorially depicted in Figure 9.6. The different terms used in this diagram are explained as follows:

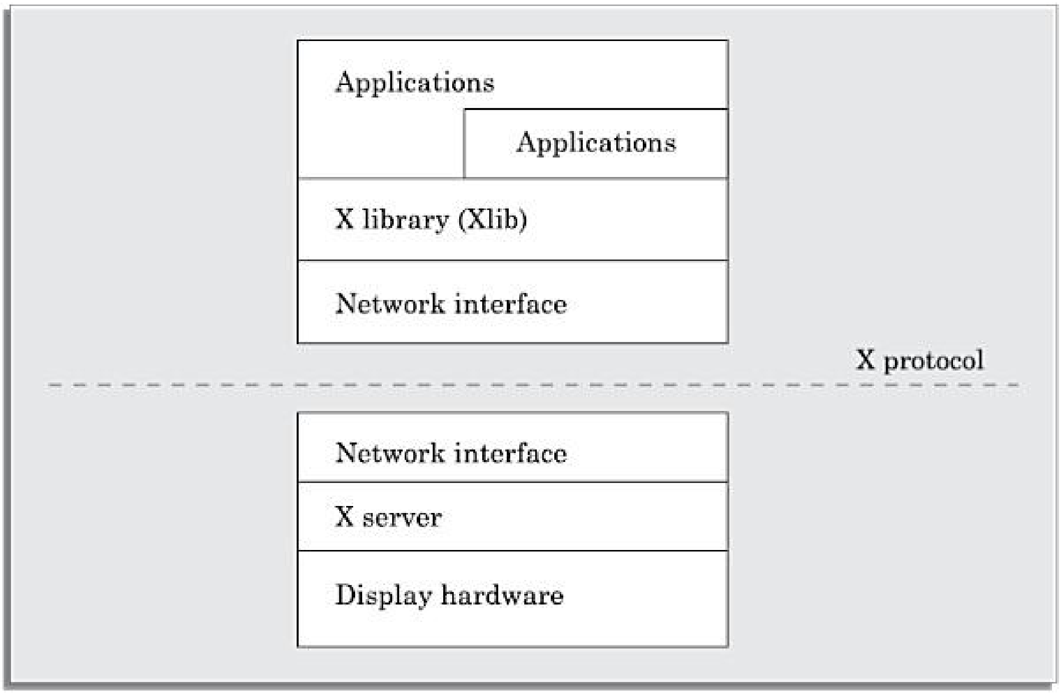


Figure 9.6: Architecture of the X System.

Xserver: The X server runs on the hardware to which the display and the key board are attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.

X protocol. The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.

X library (Xlib). The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low level primitives for developing an user interface, such as displaying a window, drawing characters and graphics on the window, waiting for specific events, etc.

Xtoolkit (Xt). The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, menu bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behaviour of these widgets by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

9.4.5 Size Measurement of a Component-based GUI

Lines of code (LOC) is not an appropriate metric to estimate and measure the size of a component-based GUI. This is because, the interface is developed by integrating several pre- built components. The different components making up an interface might have been in written using code of drastically different sizes. However, as far as the effort of the GUI developer who develops an interface by integrating the components may not be affected by the code size of the components he integrates.

A way to measure the size of a modern user interface is widget points (wp). The size of a user interface (in wp units) is simply the total number of widgets used in the interface. The size of an interface in wp units is a measure of the intricacy of the interface and is more or less independent of the implementation environment. The wp measure opens up chances for contracts on a measured amount of user interface functionality, instead of a vague definition of a complete system. However, till now there is no reported results to estimate the development effort in terms of the wp metric. An alternate way to compute the size of GUI is to simply count the number of screens. However, this would be inaccurate since a screen complexity can range from very simple to very complex.

9.5 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.

9.5.1 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 task 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.

9.5.2 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 in

Figure 9.7.

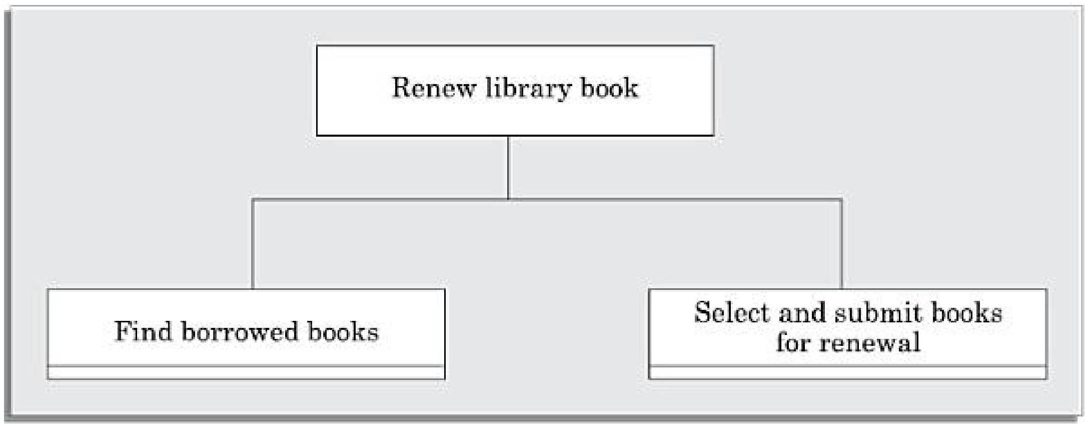


Figure 9.7: Decomposition of a task into subtasks.

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.

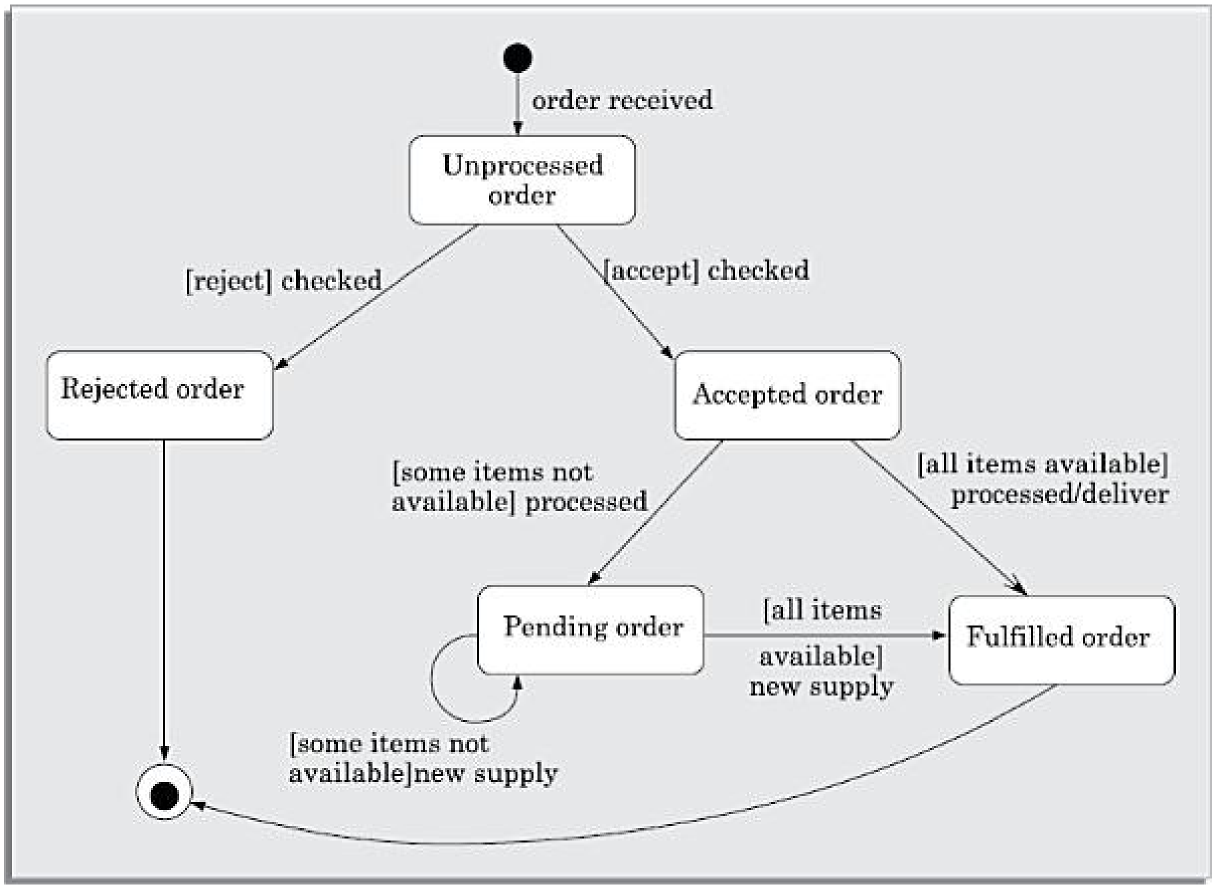

order received
Unprocessed order
[reject] checked
[accept] checked
Rejected order
Accepted order
[some items not available] processed
[all items available] processed/deliver
Pending order
[all items available] new supply
Fulfilled order
[some items not available]new supply

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.

Example 9.1 We need to develop the interface for a web-based pay-order shop, where the users can examine the contents of the shop through a web browser and can order them.

Several metaphors are possible for different parts of this problem as follows:

- Different items can be picked up from racks and examined. The user can request for the **catalog** associated with the items by clicking on the item.
- Related items can be picked from the drawers of an item cabinet.
- The items can be organised in the form of a book, similar to the way information abo u t electronic components are organised in a semiconductor hand book.

Once the users make up their mind about an item they wish to buy, they can put them into a shopping cart.

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.

saving becomes possible. According to experience reports, well-established object-oriented development environment can help to reduce development costs by as much as 20 per cent to 50 per cent over a traditional development environment.

Disadvantages of OOD

The following are some of the prominent disadvantages inherent to the object paradigm:

- The principles of abstraction, data hiding, inheritance, etc. do incur run time overhead due to the additional code that gets generated on account of these features. This causes an project-oriented program to run a little slower than an equivalent procedural program.
- An important consequence of object-orientation is that the data that is centralised in a procedural implementation, gets scattered across various objects in an object-oriented implementation. Therefore, the spatial locality of data becomes weak and this leads to higher cache miss ratios and consequently to larger memory access times. This finally shows up as increased program run time.

As we can see, increased run time is the principal disadvantage of object-orientation and higher productivity is the major advantage. In the present times, computers have become remarkably fast, and a small run time overhead is not an issue at all. Consequently, the advantages of OOD overshadow the disadvantages.

7.2 UNIFIED MODELLING LANGUAGE (UML)

As the name itself implies, UML is a language for documenting models. As is the case with any other language, UML has its syntax (a set of basic symbols and sentence formation rules) and semantics (meanings of basic symbols and sentences). It provides a set of basic graphical notations (e.g. rectangles, lines, ellipses, etc.) that can be combined in certain ways to document the design and analysis results.

It is important to remember that UML is neither a system design or development methodology by itself, nor is tied to any specific methodology. UML is merely a language for documenting models. Before the advent of UML, every design methodology not only prescribed entirely different design steps, but each was tied to some specific design modelling language. For example, OMT methodology had its own design methodology and had its own unique set of notations. So was the case with Booch's methodology, and so on. This situation made it hard for someone familiar with one methodology to understand the design solutions developed and documented using another methodology. In general, reuse of design solutions across different methodologies was hard. UML was intended to address this problem that was inherent to the modelling techniques that existed.

UML can be used to document object-oriented analysis and design results that have been obtained using any methodology.

One of the objectives of the developers of UML was to keep the notations of UML independent of any specific design methodology, so that it can be used along with any specific design methodology. In this respect, UML is different from its predecessors (e.g., OMT, Booch's methodology, etc.) where the notations supported by the modelling languages were closely tied to the corresponding design methodologies.

7.2.1 Origin of UML

In the late eighties and early nineties, there was a proliferation of object-oriented design techniques and notations. Many of these had become extremely popular and were widely used. However, the notations they used and the specific design paradigms that they advocated, differed from each other in major ways. With so many popular techniques to choose from, it was not very uncommon to find different project teams in the same organisation using different methodologies and documenting their object-oriented analysis and design results using different notations. These diverse notations used for documenting design solutions gave arise to a lot of confusion among the team members and made it extremely difficult to reuse designs across projects and communicating ideas across project teams.

UML was developed to standardise the large number of object-oriented modelling notations that existed in the early nineties. The principal ones in use those days include the following:

- OMT [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- OOSE [Jacobson 1992]
- Odell's methodology [Odell 1992]

- Shlaer and Mellor methodology[Shlaer 1992]

Needless to say that UML has borrowed many concepts from these modeling techniques. Concepts and notations from especially the first three methodologies have heavily been drawn upon. The influence of various object modeling techniques on UML is shown schematically in Figure 7.12. As shown in Figure 7.12, OMT had the most profound influence on UML.

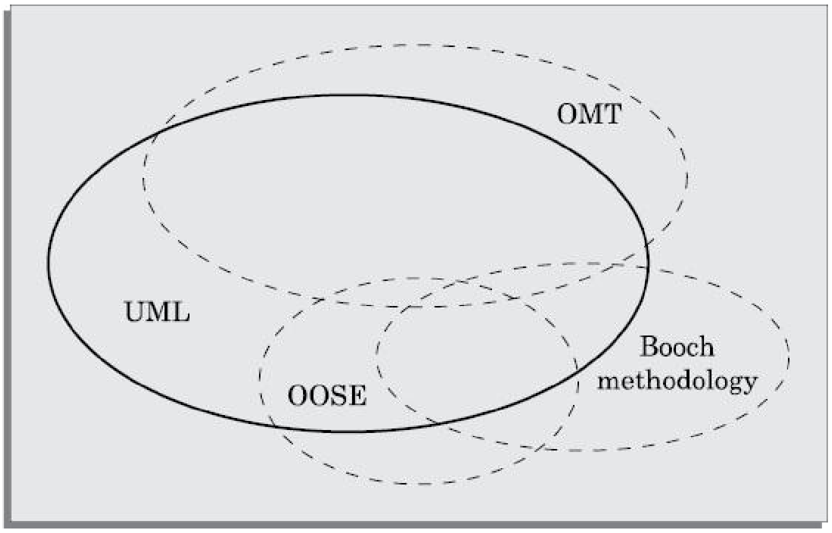


Figure 7.12: Schematic representation of the impact of different object modelling techniques on UML.

UML was adopted by object management group (OMG) as a de facto standard in 1997. Actually, OMG is not a standards formulating body, but is an association of industries that tries to facilitate early formulation of standards. OMG aims to promote consensus notations and techniques with the hope that if the usage becomes wide-spread, then they would automatically become standards. For more information on OMG, see www.omg.org. With widespread use of UML, ISO adopted UML a standard (ISO 19805) in 2005, and with this UML has become an official standard; this has further enhanced the use of UML.

UML is more complex than its antecedents. This is only natural and expected because it is intended to be more comprehensive and applicable to a wider gamut of problems than any of the notations that existed before UML. UML contains an extensive set of notations to help document several aspects (views) of a design solution through many types of diagrams. UML has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and subsequently by ISO as well as a strong industry backing have helped UML to find wide spread acceptance. UML is now being used in academic and research institutions as well as in large number of software development projects world-wide. It is interesting to note that the use of UML is not restricted to the software industry alone. As an example of UML's use outside the software development problems, some car manufacturers are planning to use UML for their "build-to-order" initiative.

Many of the UML notations are difficult to draw by hand on a paper and are best drawn using a CASE tool such as Rational Rose© (see www.rational.com) or MagicDraw (www.magicdraw.com). Now several free UML CASE tools are also available on the web. Most of the available CASE tools help to refine an initial object model to final design, and these also automatically generate code templates in a variety of languages, once the UML models have been constructed.

7.2.2 Evolution of UML

Since the release of UML 1.0 in 1997, UML continues to evolve (see Figure 7.13) with feedback from practitioners and academicians to make it applicable to different system development situations. Almost every year several new releases (shown as UML 1.X in Figure 7.13) were announced. A major milestone in the evolution of UML was the release of UML 2.0 in the year 2007. Since the use of embedded applications is increasing rapidly, there was popular demand to extend UML to support the special concepts and notations required to develop embedded applications. UML 2.0 was an attempt to make UML applicable to the development of concurrent and embedded systems. For this, many new features such as events, ports, and frames in sequence diagrams were introduced. We briefly discuss these developments in this chapter.

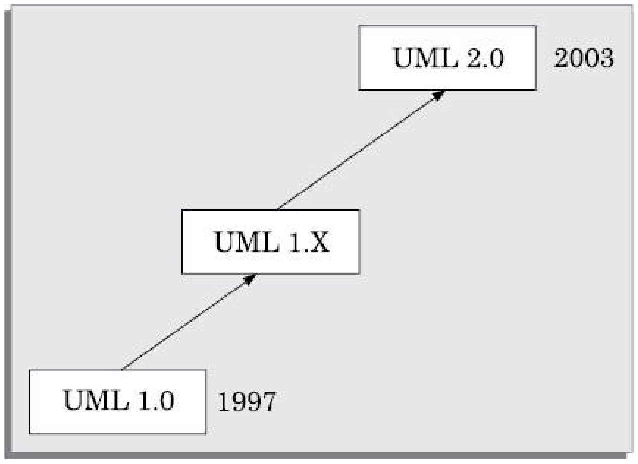


Figure 7.13: Evolution of UML.

What is a model?

Before we discuss the features of UML in detail, it is important to understand what exactly is meant by a model, and why is it necessary to create a model.

A model is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details. This reduces the problem complexity and makes it easy to understand the problem (or situation).

A model is a simplified version of a real system. It is useful to think of a model as capturing aspects important for some application while omitting (or abstracting out) the rest. As we had already pointed out in Chapter 1, as the size of a problem increases, the perceived complexity increases exponentially due to human cognitive limitations. Therefore, to develop a good understanding of any problem, it is necessary to construct a model of the problem. Modelling has turned out to be a very essential tool in software design and helps to effectively handle the complexity in a problem. These models that are first constructed are the models of the problem. A design methodology essentially transform these analysis models into a design model through iterative refinements.

Different types of models are obtained based on the specific aspects of the actual system that are ignored while constructing the model. To understand this, let us consider the models constructed by an architect of a large building. While constructing the frontal view of a large building (elevation plan), the architect ignores aspects such as floor plan, strength of the walls, details of the inside architecture, etc. While constructing the floor plan, he completely ignores the frontal view (elevation plan), site plan, thermal and lighting characteristics, etc. of the building.

A model in the context of software development can be graphical, textual, mathematical, or program code-based. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, there are certain modelling situations (discussed later in this Chapter), for which in addition to the graphical UML models, separate textual explanations are required to accompany the graphical models.

Why construct a model?

An important reason behind constructing a model is that it helps to manage the complexity in a problem and facilitates arriving at good solutions and at the same time helps to reduce the design costs. The initial model of a problem is called an analysis model. The analysis model of a problem can be refined into a design model using a design methodology. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Design
- Coding
- Visualisation and understanding of an implementation.
- Testing, etc.

Since a model can be used for a variety of purposes, it is reasonable to expect that the models would vary in detail depending on the purpose for which these are being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is constructed for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model constructed for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed.

We now discuss the different types of UML diagrams and the notations used to develop these diagrams.

7.3 UML DIAGRAMS

In this section, we discuss the diagrams supported by UML 1.0. Later in Section 7.9.2, we discuss the changes to UML 1.0 brought about by UML 2.0. UML 1.0 can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modelled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of a software system to be developed and facilitate a comprehensive understanding of the system. Each perspective focuses on some specific aspect and ignores the rest. Some may ask, why construct several models from different perspectives—why not just construct one model that captures all perspectives? The answer to this is the following:

If a single model is made to capture all the required perspectives, then it would be as complex as the original problem, and would be of little use.

Once a system has been modelled from all the required perspectives, the constructed models can be refined to get the actual implementation of the system.

UML diagrams can capture the following views (models) of a system:

- User's view
- Structural view
- Behaviourial view
- Implementation view
- Environmental view

Figure 7.14 shows the different views that the UML diagrams can document. Observe that the users' view is shown as the central view. This is because based on the users' view, all other views are developed and all views need to conform to the user's view. Most of the object oriented analysis and design methodologies, including the one we are going to discuss in Chapter 8 require us to iterate among the different views several times to arrive at the final design. We first provide a brief overview of the different views of a system which can be documented using UML. In the subsequent sections, the diagrams used to realize the important views are discussed.

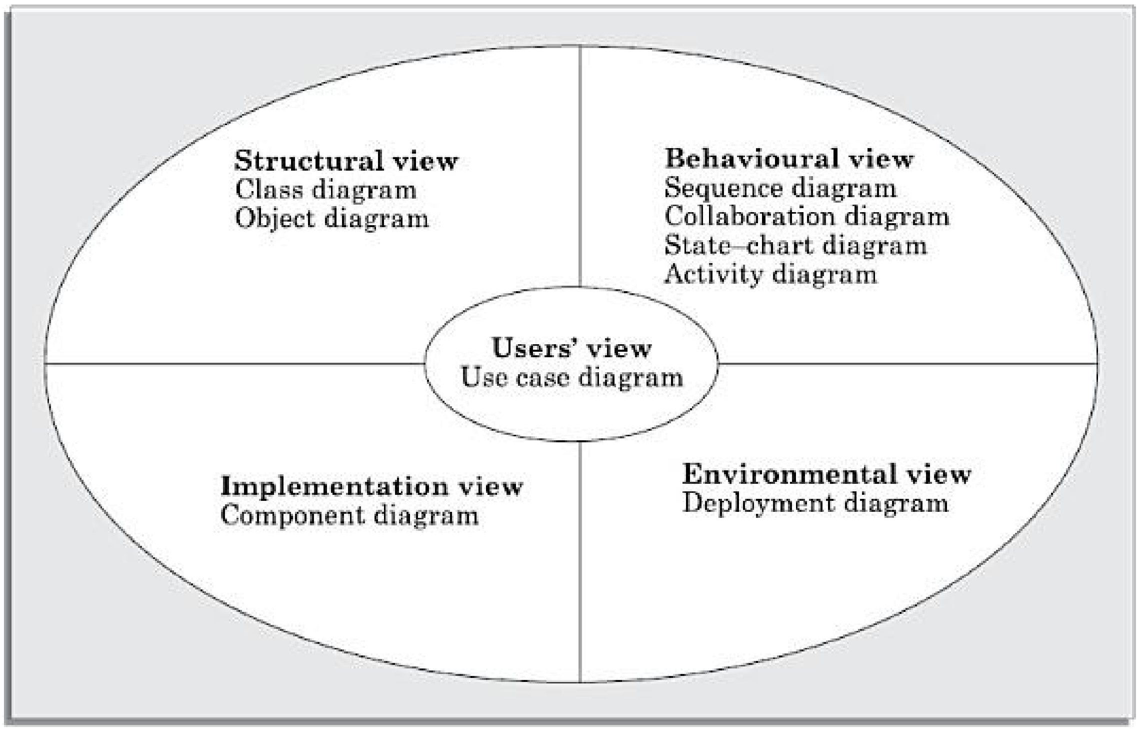


Figure 7.14: Different types of diagrams and views supported in UML.

Users' view

This view defines the functionalities made available by the system to its users.

The users' view captures the view of the system in terms of the functionalities offered by the system to its users.

The users' view is a black-box view of the system where the internal structure, the dynamic behaviour of different system components, the implementation etc. are not captured. The users' view is very different from all other views in the sense that it is a functional model[1] compared to all other views that are essentially object models.[2]

The users' view can be considered as the central view and all other views are required to conform to this view. This thinking is in fact the crux of any user centric development style. It is indeed remarkable that even for object-oriented development, we need a functional view. That is because, after all, a user considers a system as providing a set of functionalities.

Structural view

The structural view defines the structure of the problem (or the solution) in terms of the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects).

The structural model is also called the static model, since the structure of a system does not change with time.

Behaviourial view

The behaviourial view captures how objects interact with each other in time to realise the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system. It therefore constitutes the dynamic model of the system.

Implementation view

This view captures the important components of the system and their interdependencies. For example, the implementation view might show the GUI part, the middleware, and the database part as the different parts and also would capture their interdependencies.

Environmental view

This view models how the different components are implemented on different pieces of hardware.

For any given problem, should one construct all the views using all the diagrams provided by UML? The answer is No. For a simple system, the use case model, class diagram, and one of the interaction diagrams may be sufficient. For a system in which the objects undergo many state changes, a state chart diagram may be necessary. For a system, which is implemented on a large number of hardware components, a deployment diagram may be necessary. So, the type of models to be constructed depends on the problem at hand. Rosenberg provides an analogy [Ros 2000] saying that "Just like you do not use all the words listed in the dictionary while writing a prose, you do not use all the UML diagrams and modeling elements while modeling a system."

7.4 USE CASE MODEL

The use case model for any system consists of a set of use cases.

Intuitively, the use cases represent the different ways in which a system can be used by the users.

A simple way to find all the use cases of a system is to ask the question —"What all can the different categories of users do by using the system?" Thus, for the library information system (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc.

Roughly speaking, the use cases correspond to the high-level functional requirements that we discussed in Chapter 4. We can also say that the use cases partition the system behaviour into transactions, such that each transaction performs some useful action from the user's point of view. Each transaction, to complete, may involve multiple message exchanges between the user and the system.

The purpose of a use case is to define a piece of coherent behaviour without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software. A use case typically involves a sequence of interactions between the user and the system. Even for the same use case, there can be several different sequences of interactions. A use case consists of one main line sequence and several alternate sequences. The main line sequence represents the interactions between a user and the system that normally take place. The mainline sequence is the most frequently occurring sequence of interaction. For example, in the mainline sequence of the withdraw cash use case supported by a bank ATM would be—the user inserts the ATM card, enters password, selects the amount withdraw option, enters the amount to be withdrawn, completes the transaction, and collects the amount. Several variations to the main line sequence (called alternate sequences) may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, consider the following variations or alternate sequences:

- Password is invalid.
- The amount to be withdrawn exceeds the account balance.

The mainline sequence and each of the alternate sequences corresponding to the invocation of a use case is called a scenario of the use case.

A use case can be viewed as a set of related scenarios tied together by a common goal. The main line sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity.

Normally, each use case is independent of the other use cases. However, implicit dependencies among use cases may exist because of dependencies that may exist among use cases at the implementation level due to factors such as shared resources, objects, or functions. For example, in the Library Automation System example, `renew-book` and `reserve-book` are two independent use cases. But, in actual implementation of renew-book, a check is to be made to see if any book has been reserved by a previous execution of the `reserve-book` use case. Another example of dependence among use cases is the following. In the Bookshop Automation Software, `update-inventory` and `sale-book` are two independent use cases. But, during execution of sale-book there is an implicit dependency on `update-inventory`. Since when sufficient quantity is unavailable in the inventory, `sale-book` cannot operate until the inventory is replenished using `update-inventory`.

The use case model is an important analysis and design artifact. As already mentioned, other UML models must conform to this model in any use case-driven (also called as the user-centric) analysis and development approach. It should be remembered that the "use case model" is not really an object-oriented model according to a strict definition of the term.

In contrast to all other types of UML diagrams, the use case model represents a functional or process model of a system.

7.4.1 Representation of Use Cases

A use case model can be documented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (e.g., library information system) appears inside the rectangle.

The different users of the system are represented by using stick person icons. Each stick person icon is referred to as an actor.[3] An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. An actor can participate in one or more use cases. The line connecting an actor and the use case is called the communication relationship. It indicates that an actor makes use of the functionality provided by the use case.

Both human users and external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

At this point, it is necessary to explain the concept of a stereotype in UML. One of the main objectives of the creators of the UML was to restrict the number of primitive symbols in the language. It was clear to them that when a language has a large number of primitive symbols, it becomes very difficult to learn use. To convince yourself, consider that English with 26 alphabets is much easier to learn and use compared to the Chinese language that has thousands of symbols. In this context, the primary objective of stereotype is to reduce the number of different types of symbols that one needs to learn.

The stereotype construct when used to annotate a basic symbol, can give slightly different meaning to the basic symbol— thereby eliminating the need to have several symbols whose meanings differ slightly from each other.

Just as you stereotype your friends as studious, jovial, serious, etc. stereotyping can be used to give special meaning to any basic UML construct. We shall, later on, see how other UML constructs can be stereotyped. We can stereotype the stick person icon symbol to denote an external system. If the developers of UML had assigned a separate symbol to denote things such as an external system, then the number of basic symbols one would have to learn and remember while using UML would have increased significantly. This would have certainly made learning and using UML much more difficult.

You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not. However, drawing the system boundary is optional.

We now give a few examples to illustrate how use cases of a system can be documented.

Example 7.2 The use case model for the Tic-tac-toe game software is shown in Figure 7.15. This software has only one use case, namely, "play move". Note that we did not name the use case "get-user-move", as "get-user-move" would be inappropriate because this would represent the developer's perspective of the use case. The use cases should be named from the users' perspective.

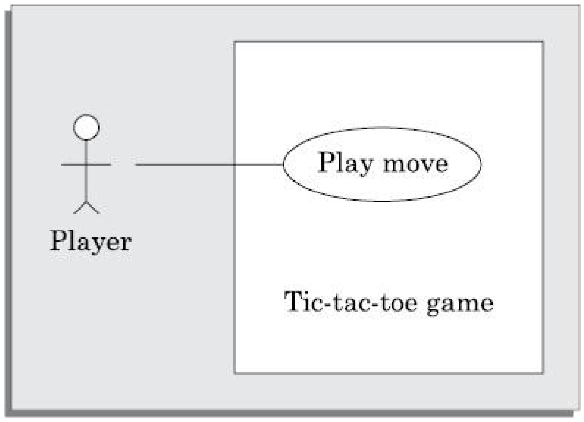


Figure 7.15: Use case model for Example 7.2.

Text description

Each ellipse in a use case diagram, by itself conveys very little information, other than giving a hazy idea about the use case. Therefore, every use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer as well as other relevant aspects of the use case. It should include all the behaviour associated with the use case in terms of the mainline sequence, various alternate sequences, the system responses associated with the use case, the exceptional conditions that may occur in the behaviour, etc. The behaviour description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is helpful. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternate scenarios.

Contact persons: This section lists of personnel of the client organisation with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using a use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requiremen t s : This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requiremen t s : These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation.

Example 7.3 The use case diagram of the Super market prize scheme described in example 6.3 is shown in Figure 7.16.

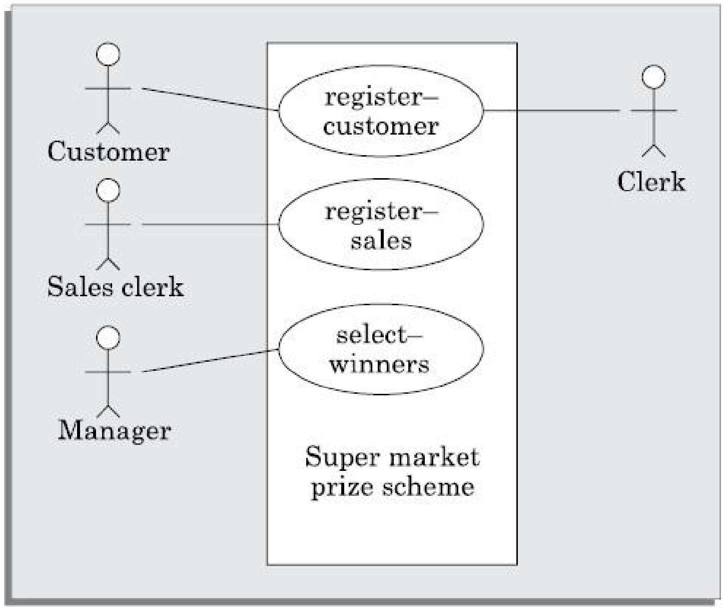


Figure 7.16: Use case model for Example 7.3.

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. Customer: select register customer option
2. System: display prompt to enter name, address, and telephone number.
3. Customer: enter the necessary values
4: System: display the generated id and the message that the customer has successfully been registered.

Scenario 2: At step 4 of mainline sequence

4: System: displays the message that the customer has already registered.

Scenario 3: At step 4 of mainline sequence

4: System: displays message that some input information have not been entered. The system displays a prompt to enter the missing values.

U2: register-sales: Using this use case, the clerk can register the details of the purchase made by a customer.

Scenario 1: Mainline sequence

1. Clerk: selects the register sales option.

2. System: displays prompt to enter the purchase details and the id of the customer.
3. Clerk: enters the required details.
4 : System: displays a message of having successfully registered the sale.

U3: select-winners. Using this use case, the manager can generate the winner list.

Scenario 2: Mainline sequence

1. Manager: selects the select-winner option.
2. System: displays the gold coin and the surprise gift winner list.

7.4.2 Why Develop the Use Case Diagram?

If you examine a use case diagram, the utility of the use cases represented by the ellipses would become obvious. They along with the accompanying text description serve as a type of requirements specification of the system and the model based on which all other models are developed. In other words, the use case model forms the core model to which all other models must conform. But, what about the actors (stick person icons)? What way are they useful to system development? One possible use of identifying the different types of users (actors) is in implementing a security mechanism through a login system, so that each actor can invoke only those functionalities to which he is entitled to. Another important use is in designing the user interface in the implementation of the use case targetted for each specific category of users who would use the use case. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

7.4.3 How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the high-level requirements specified in the SRS document correspond to the use cases. In the absence of a well-formulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the different functions that they might initiate or participate are identified. For example, for a Library Automation System, the categories of users can be members, librarian, and the accountant. Each user typically focuses on a set of functionalities. Foe example, the member typically concerns himself with book issue, return, and renewal aspects. The librarian concerns himself with creation and deletion of the member and book records. The accountant concerns itself with the amount collected from membership fees and the expenses aspects.

7.4.4 Essential Use Case versus Real Use Case

Essential use cases are created during early requirements elicitation. These are also early problem analysis artifacts. They are independent of the design decisions and tend to be correct over long periods of time.

Real use cases describe the functionality of the system in terms of its actual current design committed to specific input/output technologies. Therefore, the real use cases can be developed only after the design decisions have been made. Real use cases are a design artifact. However, sometimes organisations commit to development contracts that include the detailed user interface specifications. In such cases, there is no distinction between the essential use case and the real use case.

7.4.5 Factoring of Commonality among Use Cases

It is often desirable to factor use cases into component use cases. All use cases need not be factored. In fact, factoring of use cases are required under two situations as follows:

- Complex use cases need to be factored into simpler use cases. This would not only make the behaviour associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single standard-sized (A4) paper.
- Use cases need to be factored whenever there is common behaviour across different use cases. Factoring would make it possible to define such behaviour only once and reuse it wherever required.

It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it. UML offers three factoring mechanisms as discussed further.

Generalisation

Use case generalisation can be used when you have one use case that is similar to another, but does something slightly differently or something more. Generalisation works the same way with use cases as it does with classes. The child use case inherits the behaviour and meaning of the present use case. The notation is the same too (See Figure 7.17). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

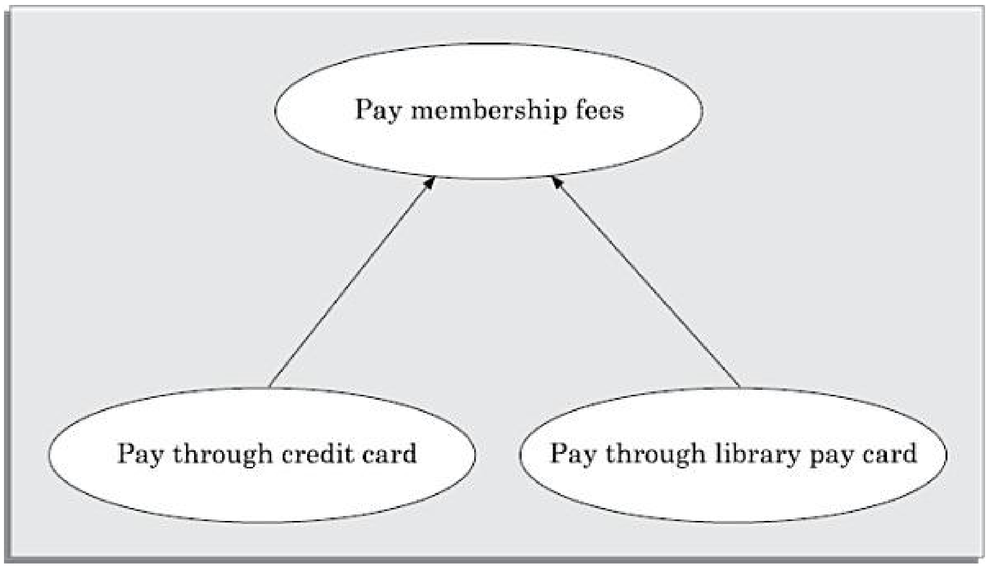


Figure 7.17: Representation of use case generalisation.

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship implies one use case includes the behaviour of another use case in its sequence of events and actions. The includes relationship is appropriate when you have a chunk of behaviour that is similar across a number of use cases. The factoring of such behaviour will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use case into more manageable parts.

As shown in Figure 7.18, the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behaviour of the common use case. As shown in example Figure 7.19, the use cases `issue-book` and `renew-book` both include `check-reservation` use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and independent text description should be provided for it.

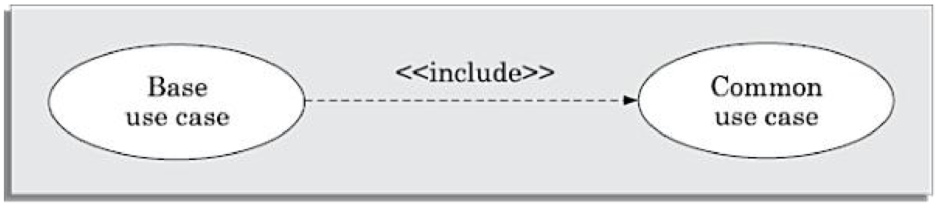


Figure 7.18: Representation of use case inclusion.

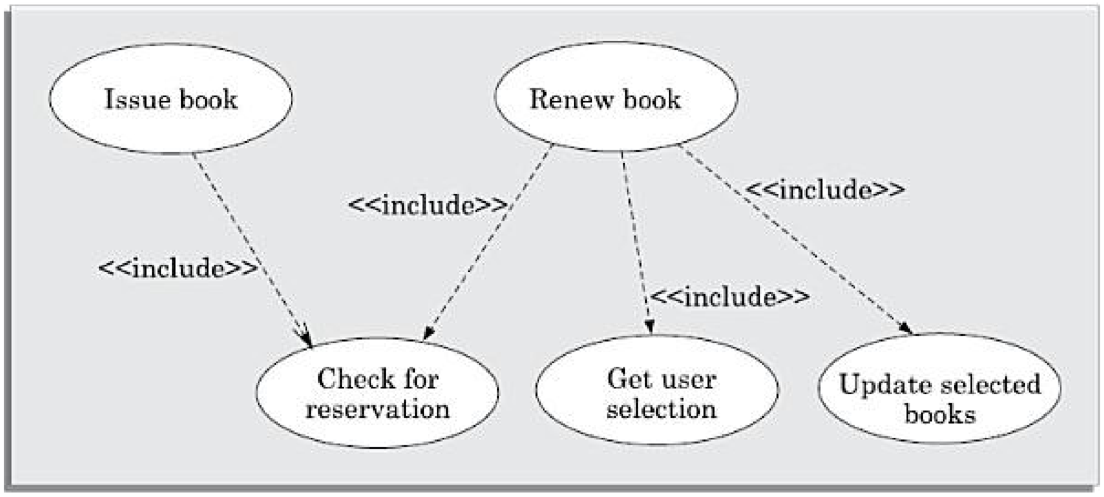


Figure 7.19: Example of use case inclusion.

Extends

The main idea behind the extends relationship among use cases is that it allows you show optional system behaviour. An optional system behaviour is executed only if certain conditions hold, otherwise the optional behaviour is not executed. This relationship among use cases is also predefined as a stereotype as shown in Figure 7.20.

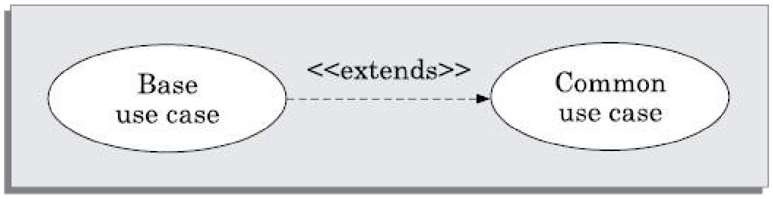


Figure 7.20: Example of use case extension.

The extends relationship is similar to generalisation. But unlike generalisation, the extending use case can add additional behaviour only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

Organisation

When the use cases are factored, they are organised hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in Figure 7.21. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organised in a hierarchy. It is not necessary to decompose the simple use cases.

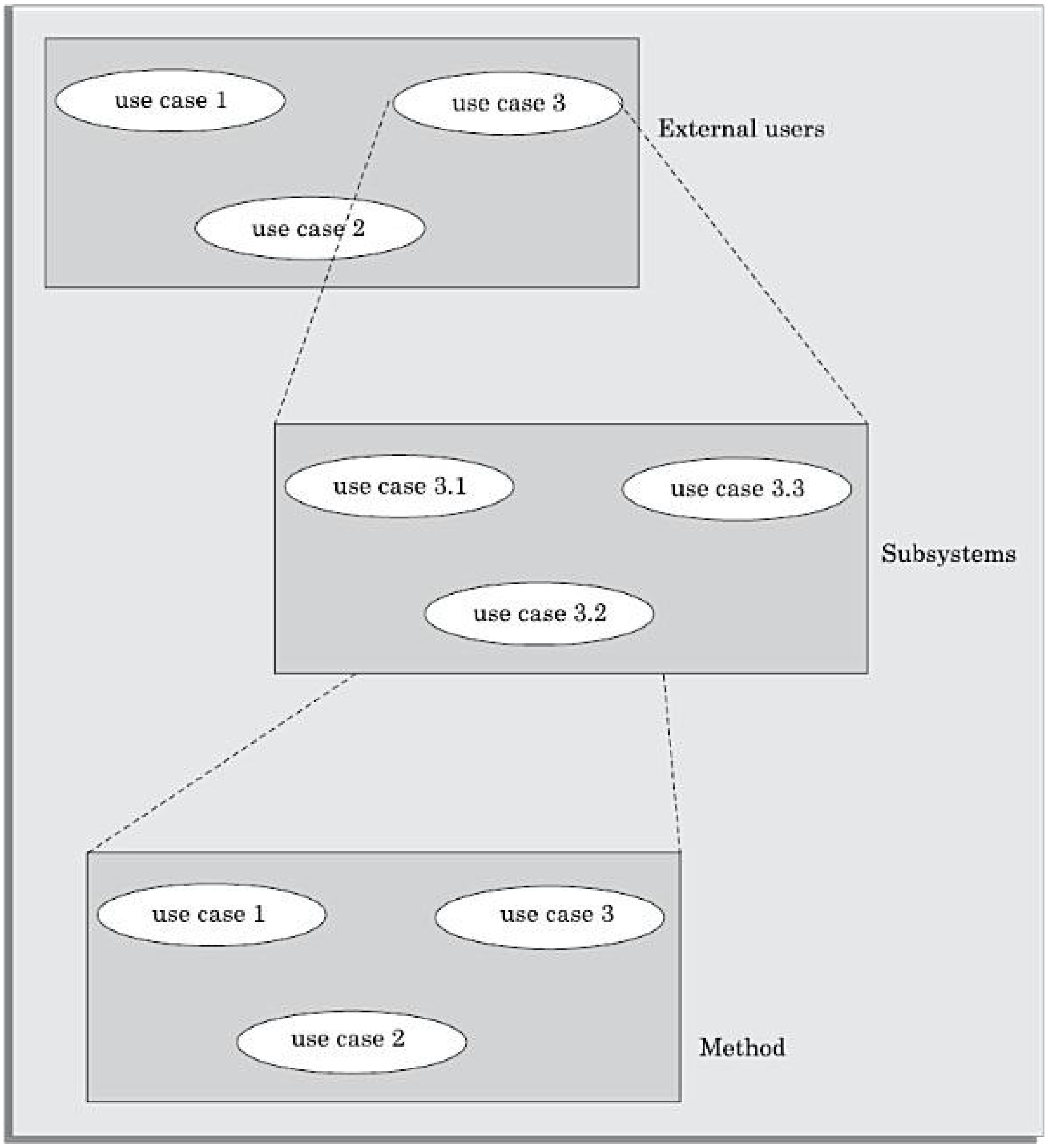


Figure 7.21: Hierarchical organisation of use cases.

The functionality of a super-ordinate use case is traceable to its subordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases.

At the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasised. In the top-level diagram, only those use cases with which external users interact are shown. The topmost use cases specify the complete services offered by the system to the external users of the system. The subsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

7.4.6 USE CASE PACKAGING

Packaging is the mechanism provided by UML to handle complexity. When we have too many use cases in the top-level diagram, we can package the related use cases so that at best 6 or 7 packages are present at the top level diagram. Any modeling element that becomes large and complex can be broken up into packages. Please note that you can put any element of UML (including another package) in a package diagram. The symbol for a package is a folder. Just as you organise a large collection of documents in a folder, you organise UML elements into packages. An example of packaging use cases is shown in Figure 7.22.

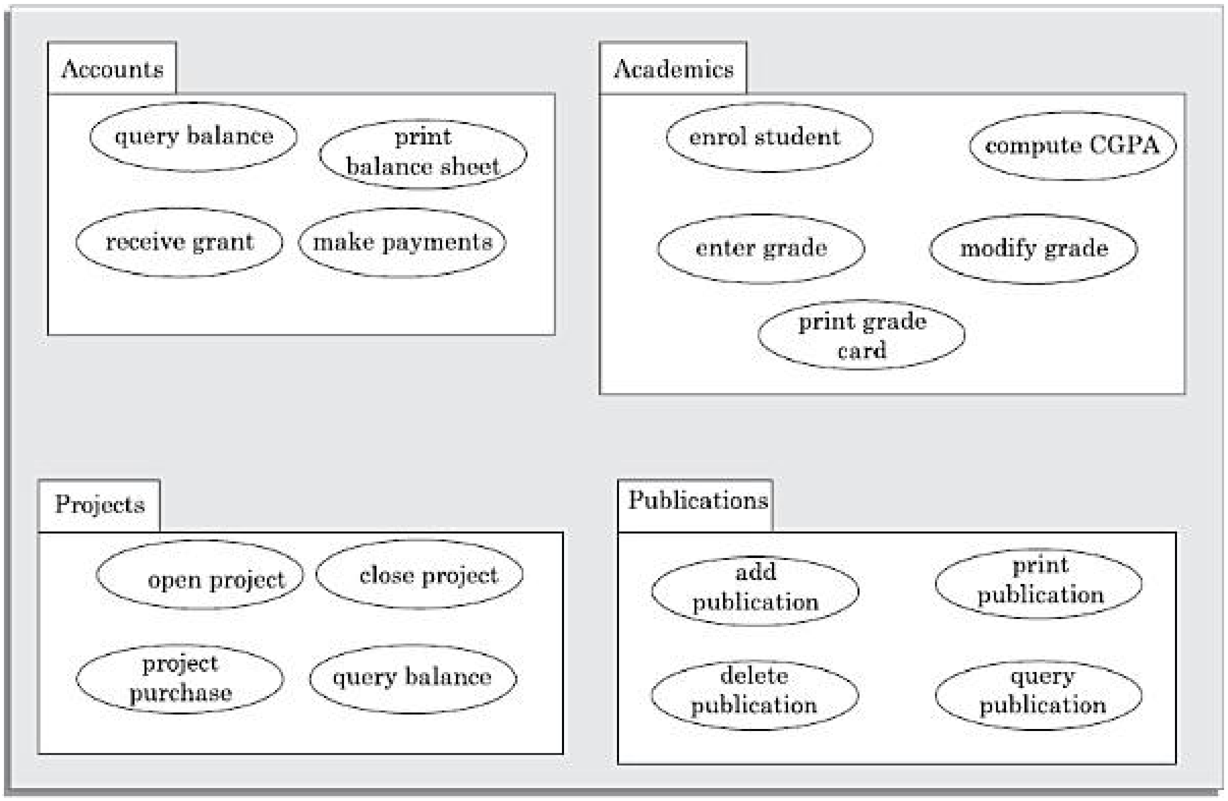


Figure 7.22: Use case packaging.

7.5 CLASS DIAGRAMS

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships—generalisation, aggregation, association, and various kinds of dependencies. We now discuss the UML syntax for representation of the classes and their relationships.

Classes

The classes represent entities with common features, i.e., attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase (e.g. `LibraryMember`). Object names on the other hand, are written using a mixed case convention, but starts with a small case letter (e.g., `studentMember`). Class names are usually chosen to be singular nouns. An example of various representations of a class are shown in Figure 7.23.

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram. But, one may wonder why there are so many representations for a class! The answer is that these different notations are used depending on the amount of information about a class is available. At the start of the design process, only the names of the classes is identified. This is the most abstract representation for the class. Later in the design process the methods for the class and the attributes are identified and the other more concrete notations are used.

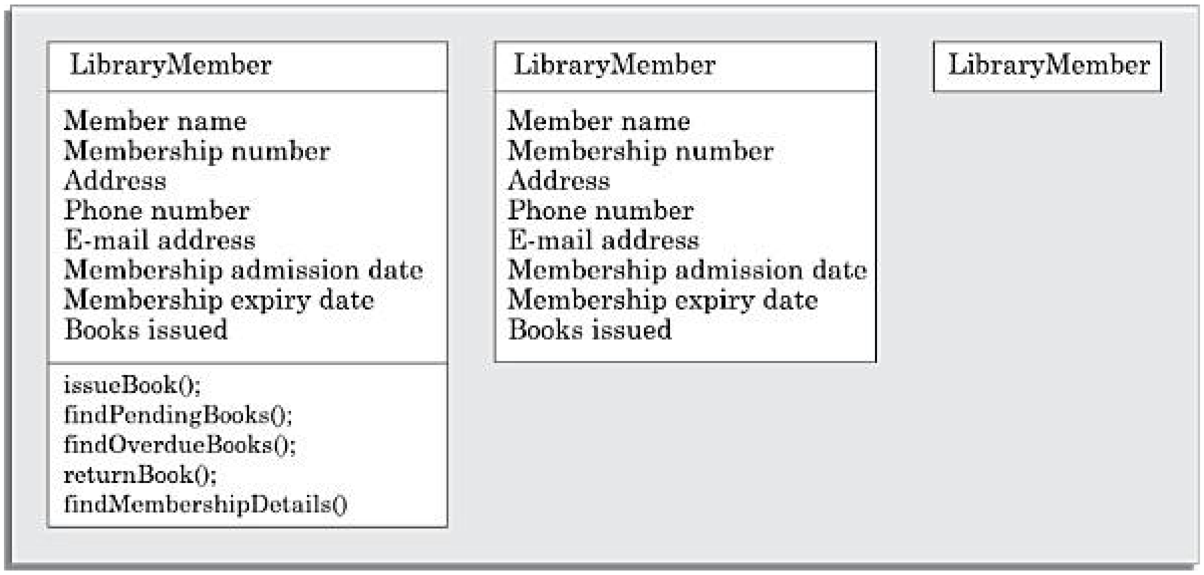


Figure 7.23: Different representations of the LibraryMember class.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type (that is, their class, e.g., Int, Book, Employee, etc.), an initial value, and constraints. Attribute names are written left-justified using plain type letters, and the names should begin with a lower case letter.

Attribute names may be followed by square brackets containing a multiplicity expression, e.g. sensorStatus[10]. The multiplicity expression indicates the number of attributes per instance of the class. An attribute without square brackets must hold exactly one value. The type of an attribute is written by following the attribute name with a colon and the type name, (e.g., sensorStatus[1]:Int).

The attribute name may be followed by an initialisation expression. The initialisation expression can consist of an equal sign and an initial value that is used to initialise the attributes of the newly created objects, e.g. sensorStatus[1]:Int=0.

Operation: The operation names are typically left justified, in plain type, and always begin with a lower case letter. Abstract operations are written in italics.[4] (Remember that abstract operations are those for which the implementation is not provided during the class definition.) The parameters of a function may have a kind specified. The kind may be "in" indicating that the parameter is passed into the operation; or "out" indicating that the parameter is only returned from the operation; or "inout" indicating that the parameter is used for passing data into the operation and getting result from the operation. The default is "in".

An operation may have a return type consisting of a single return type expression, e.g., issueBook(in bookName):Boolean. An operation may have a class scope (i.e., shared among all the objects of the class) and is denoted by underlining the operation name.

Often a distinction is made between the terms operation and method. An operation is something that is supported by a class and invoked by objects of other classes. There can be multiple methods implementing the same operation. We have pointed out earlier that this is called static polymorphism. The method names can be the same; however, it should be possible to distinguish among the methods by examining their parameters. Thus, the terms operation and method are distinguishable only when there is polymorphism. When there is only a single method implementing an operation, the terms method and operation are indistinguishable and can be used interchangeably.

Association

Association between two classes is represented by drawing a straight line between the concerned classes. Figure 7.24 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with the other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1..5. An asterisk is used as a wild card and means many (zero or more). The association of Figure 7.24 should be read as "Many books may be borrowed by a LibraryMember". Usually, associations (and links) appear as verbs in the problem statement.

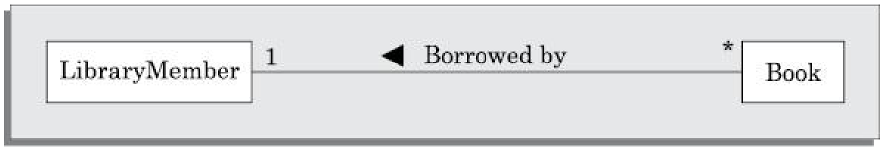


Figure 7.24: Association between two classes.

Associations are usually realised by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association relation where the involved classes are not only associated to each other, but a whole-part relationship exists between them. That is, the aggregate object not only knows the addresses of its parts and therefore invoke the methods of its parts, but also takes the responsibility of creating and destroying its parts. An example of aggregation, a book register is an aggregation of book objects. Books can be added to the register and deleted as and when required.

Aggregation is represented by an empty diamond symbol at the aggregate end of a relationship. An example of the aggregation relationship has been shown in Fig 7.25. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as aggregation of lines. Observe that the number 1 is annotated at the diamond end, and a * is annotated at the other end. This means that one document can have many paragraphs. On the other hand, if we wanted to indicate that a document consists of exactly 10 paragraphs, then we would have written number 10 in place of the (*).

The aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels. As an example of a transitive aggregation relationship, please see Figure 7.25.

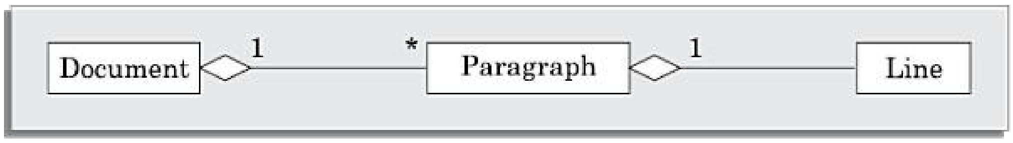


Figure 7.25: Representation of aggregation.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts cannot exist outside the whole. In other words, the lifeline of the whole and the part are identical. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

A typical example of composition is an order object where after placing the order, no item in the order cannot be changed. If any changes to any of the order items are required after the order has been placed, then the entire order has to be cancelled and a new order has to be placed with the changed items. In this case, as soon as an order object is created, all the order items in it are created and as soon as the order object is destroyed, all order items in it are also destroyed. That is, the life of the components (order items) is the same as the aggregate (order). The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in Figure 7.26.

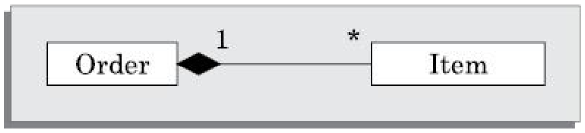


Figure 7.26: Representation of composition.

Aggregation versus Composition: Both aggregation and composition represent part/whole relationships. When the components can dynamically be added and removed from the aggregate, then the relationship is aggregation. If the components cannot be dynamically added/delete then the components are have the same life time as the composite. In this case, the relationship is represented by composition.

As an example, consider the example of an order consisting many order items. If the order once placed, the items cannot be changed at all. In this case, the order is a composition of order items. However, if order items can be changed (added, delete, and modified) after the order has been placed, then aggregation relation can be used to model it.

Inheritance

The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass. The arrow may be directly drawn from the subclass to the superclass. Alternatively, when there are many subclasses of a base class, the inheritance arrow from the subclasses may be combined to a single line (see Figure 7.27) and is labelled with the aspect of the class that is abstracted.

The direct arrows allow flexibility in laying out the diagram and can easily be drawn by hand. The combined arrows emphasise the collectivity of the subclasses, when specialisation has been done on the basis of some discriminator. In the example of Figure 7.27, issuable and reference are the discriminators. The various subclasses of a superclass can then be differentiated by means of the discriminator. The set of subclasses of a class having the same discriminator is called a partition. It is often helpful to mention the discriminator during modelling, as these become documented design decisions.

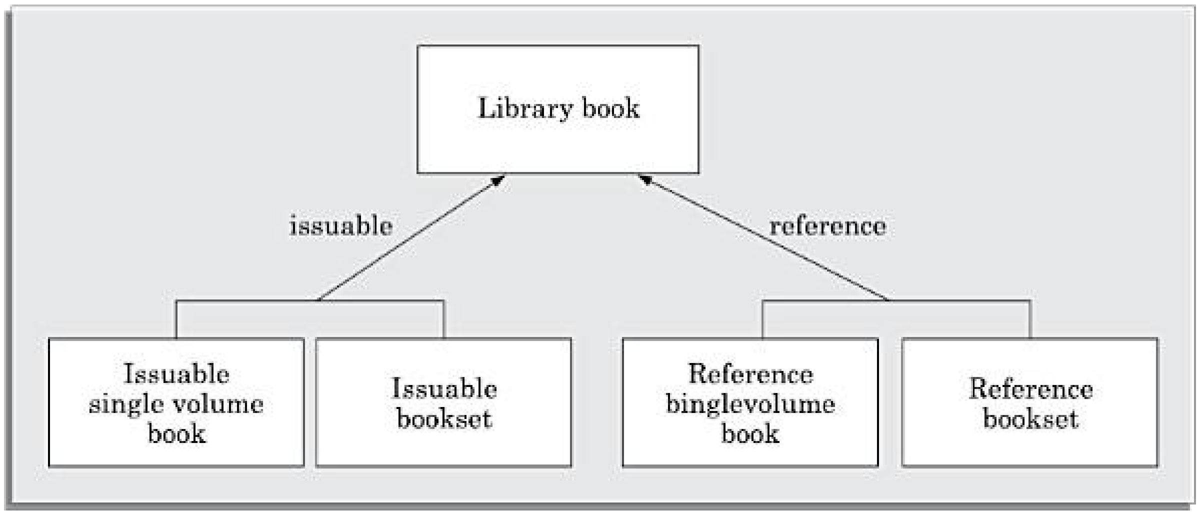


Figure 7.27: Representation of the inheritance relationship.

Dependency

A dependency relationship is shown as a dotted arrow (see Figure 7.28) that is drawn from the dependent class to the independent class.

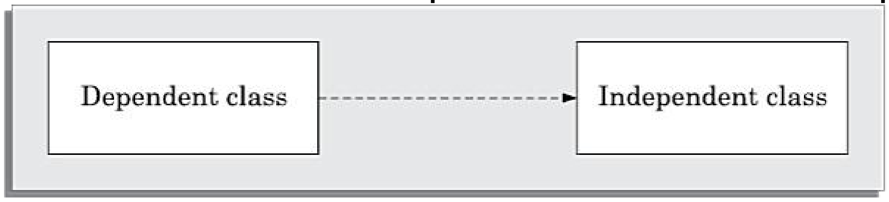


Figure 7.28: Representation of dependence between classes.

Constraints

A constraint describes a condition or an integrity rule. Constraints are typically used to describe the permissible set of values of an attribute, to specify the pre- and post-conditions for operations, to define certain ordering of items, etc. For example, to denote that the books in a library are sorted on ISBN number we can annotate the book class with the constraint

{sorted}. UML allows you to use any free form expression to describe constraints. The only rule is that they are to be enclosed within braces. Constraints can be expressed using informal English. However, UML also provides object constraint language (OCL) to specify constraints. In OCL the constraints are specified a semi-formal language, and therefore it is more amenable to automatic processing as compared to the informal constraints enclosed within {}. The interested reader is referred to [Rumbaugh1999].

Object diagrams

Object diagrams shows the snapshot of the objects in a system at a point in time. Since it shows instances of classes, rather than the classes themselves, it is often called as an instance diagram. The objects are drawn using rounded rectangles (see Figure 7.29).

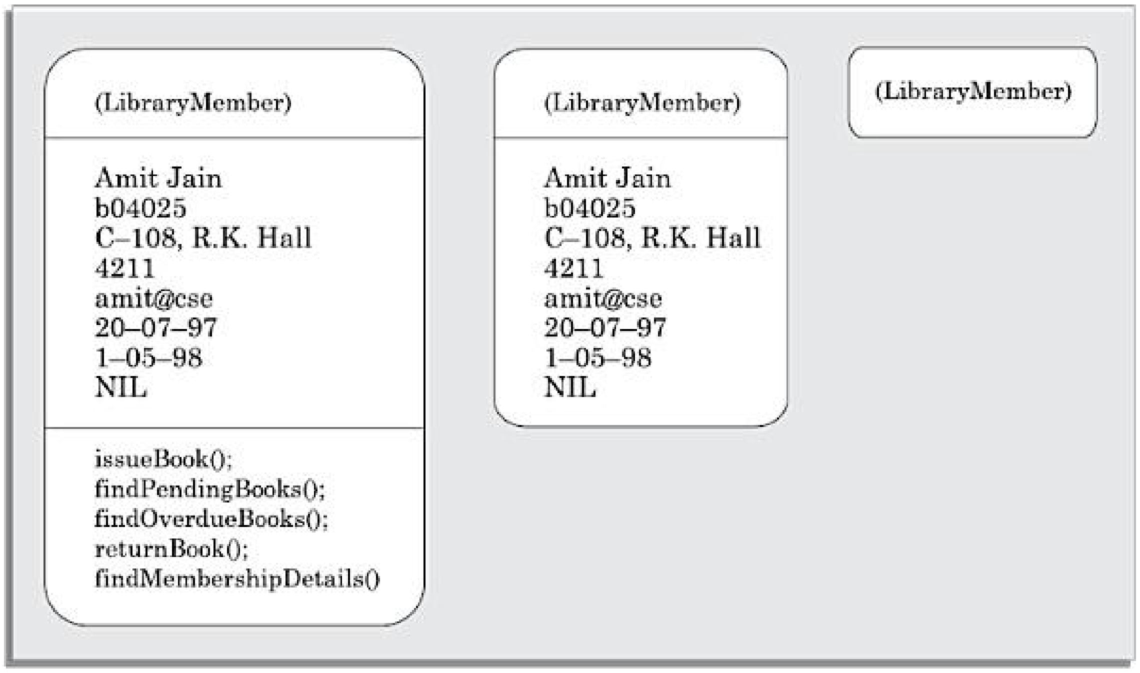


Figure 7.29: Different representations of a LibraryMember object.

An object diagram may undergo continuous change as execution proceeds. For example, links may get formed between objects and get broken. Objects may get created and destroyed, and so on. Object diagrams are useful to explain the working of a system.

7.6 INTERACTION DIAGRAMS

When a user invokes one of the functions supported by a system, the required behaviour is realised through the interaction of several objects in the system. Interaction diagrams, as their name itself implies, are models that describe how groups of objects interact among themselves through message passing to realise some behaviour.

Typically, each interaction diagram realises the behaviour of a single use case.

Sometimes, especially for complex use cases, more than one interaction diagrams may be necessary to capture the behaviour. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams—sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other. However, they are both useful. These two actually portray different perspectives of behaviour of a system and different types of inferences can be drawn from them. The interaction diagrams play a major role in any effective object-oriented design process. We discuss this issue in Chapter 8.

Sequence diagram

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined. This signifies that we are referring any arbitrary instance of the class. For example, in Figure 7.30 :Book represents any arbitrary instance of the Book class.

An object appearing at the top of the sequence diagram signifies that the object existed even before the time the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g., a method call), then the object should be shown at the appropriate place on the diagram where it is created.

The vertical dashed line is called the object’s lifeline. Any point on the lifeline implies that the object exists at that point. Absence of lifeline after some point indicates that the object ceases to exist after that point in time, particular point of time. Normally, at the point if an object is destroyed, the lifeline of the object is crossed at that point and the lifeline for the object is not drawn beyond that point. A rectangle called the activation symbol is drawn on the lifeline of an object to indicate the points of time at which the object is active. Thus an activation symbol indicates that an object is active as long as the symbol (rectangle) exists on the lifeline. Each message is indicated as an arrow between the lifelines of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur.

Each message is labelled with the message name. Some control information can also be included. Two important types of control information are:

- A condition (e.g., [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows that the message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, e.g., [for every book object].

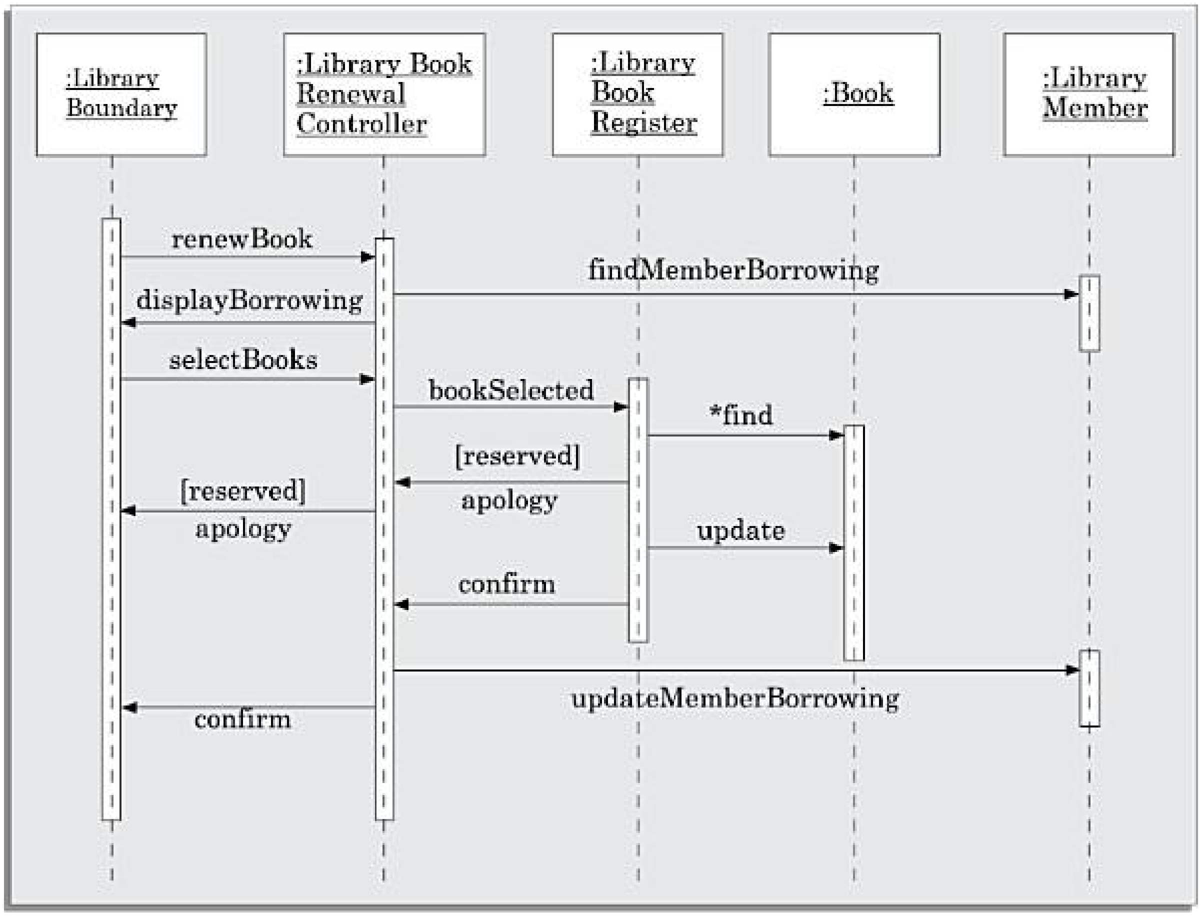


Figure 7.30: Sequence diagram for the renew book use case

The sequence diagram for the book renewal use case for the Library Automation Software is shown in Figure 7.30. Observe that the exact objects which participate to realise the renew book behaviour and the order in which they interact can be clearly inferred from the sequence diagram. The development of the sequence diagram in the development methodology (discussed in Chapter 8) would help us to determine the responsibilities that must be assigned to the different classes; i.e., what methods should be supported by each class.

Collaboration diagram

A collaboration diagram shows both structural and behavioural aspects explicitly. This is unlike a sequence diagram which shows only the behavioural aspects. The structural aspect of a collaboration diagram consists of objects and links among them indicating association. In this diagram, each object is also called a collaborator. The behavioural aspect is described by the set of messages exchanged among the different collaborators.

The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labelled arrow placed near the link. Messages are prefixed with sequence numbers because they are the only way to describe the relative sequencing of the messages in this diagram.

The collaboration diagram for the example of Figure 7.30 is shown in Figure 7.31. Use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

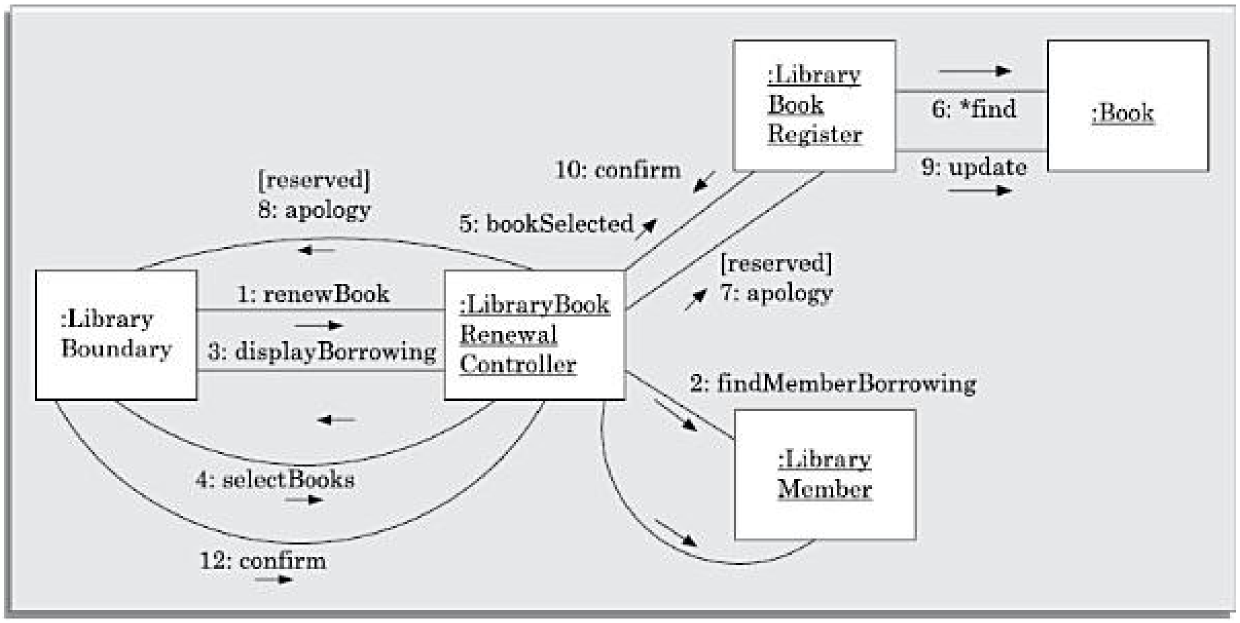


Figure 7.31: Collaboration diagram for the renew book use case.

7.7 ACTIVITY DIAGRAM

The activity diagram is possibly one modelling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It has possibly been based on the event diagram of Odell [1992] though the notation is very different from that used by Odell.

The activity diagram focuses on representing various activities or chunks of processing and their sequence of activation. The activities in general may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then exact situation under which each is executed must be identified through appropriate conditions.

Activity diagrams are similar to the procedural flow charts. The main difference is that activity diagrams support description of parallel activities and synchronisation aspects involved in different activities.

Parallel activities are represented on an activity diagram by using swim lanes. Swim lanes enable you to group activities based on who is performing them, e.g., academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lanes can be assigned to some model elements, e.g. classes or some component, etc. For example, in Figure 7.32 the swim lane corresponding to the academic section, the activities that are carried out by the academic section and the specific situation in which these are carried out are shown.

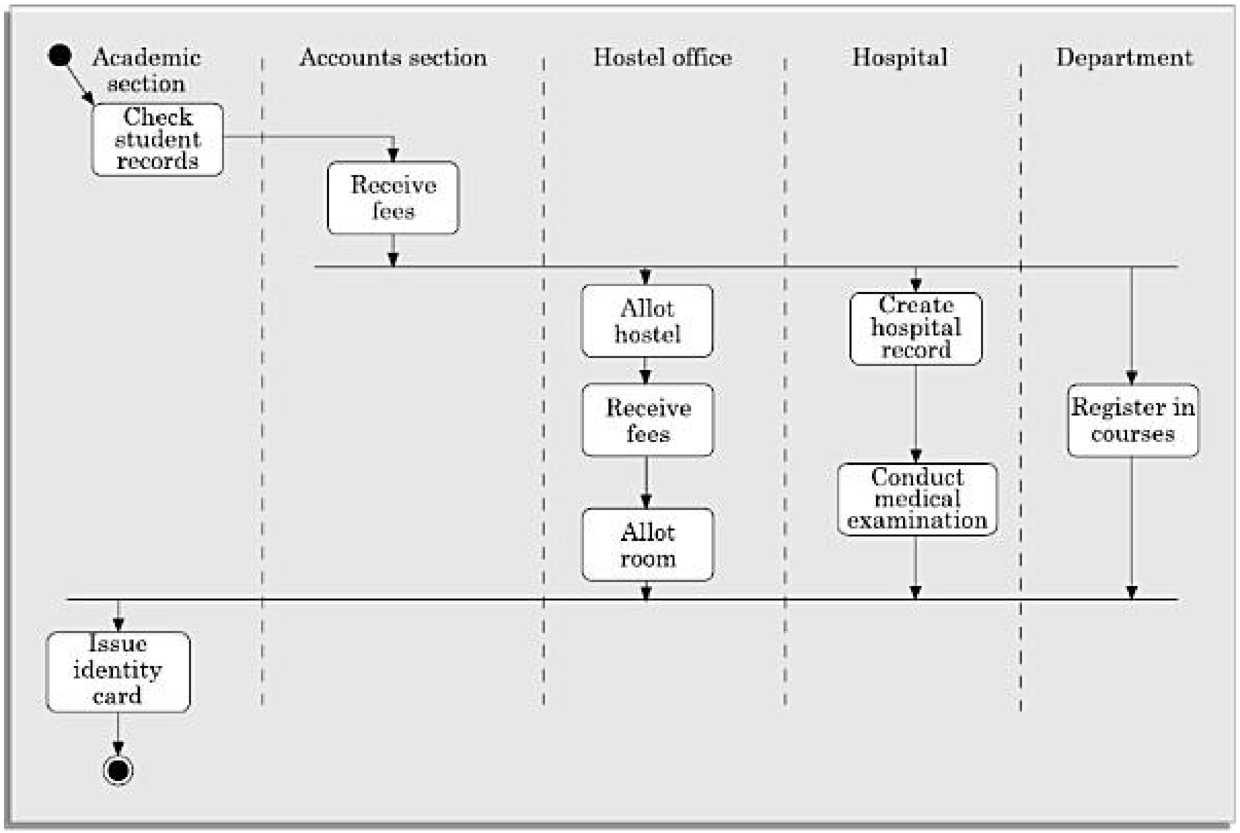


Figure 7.32: Activity diagram for student admission procedure at IIT.

Activity diagrams are normally employed in business process modelling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving the roles played by many components. Besides helping the developer to understand the complex processing activities, these diagrams can also be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

The student admission process in IIT is shown as an activity diagram in Figure 7.32. This shows the part played by different components of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this is a synchronisation issue and is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

7.8 STATE CHART DIAGRAM

A state chart diagram is normally used to model how the state of an object changes in its life time. State chart diagrams are good at describing how the behaviour of an object changes across several use case executions. However, if we are interested in modelling some behaviour that involves several objects collaborating with each other, state chart diagram is not appropriate. We have already seen that such behaviour is better modelled using sequence or collaboration diagrams. State chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the object being modelled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modelling, it has even been used in theoretical computer science as a generator for regular languages.

Why state chart?

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).

Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

Basic elements of a state chart

The basic elements of the state chart diagram are as follows:

Initial state: This represented as a filled circle.

Final state: This is represented by a filled circle inside a larger circle.

State: These are represented by rectangles with rounded corners.

Transition: A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is places along side the arrow. You can also assign a guard to the transition. A guard is a Boolean logic condition. The transition can take place only if the guard evaluates to true. The syntax for the label of the transition is shown in 3 parts—[guard]event/action.

An example state chart for the order object of the Trade House Automation software is shown in Figure 7.33. Observe that from Rejected order state, there is an automatic and implicit transition to the end state. Such transitions are called pseudo transitions.

7.9 POSTSCRIPT

UML has gained rapid acceptance among practitioners and academicians over a short time and has proved its utility in arriving at good design solutions to software development problems.

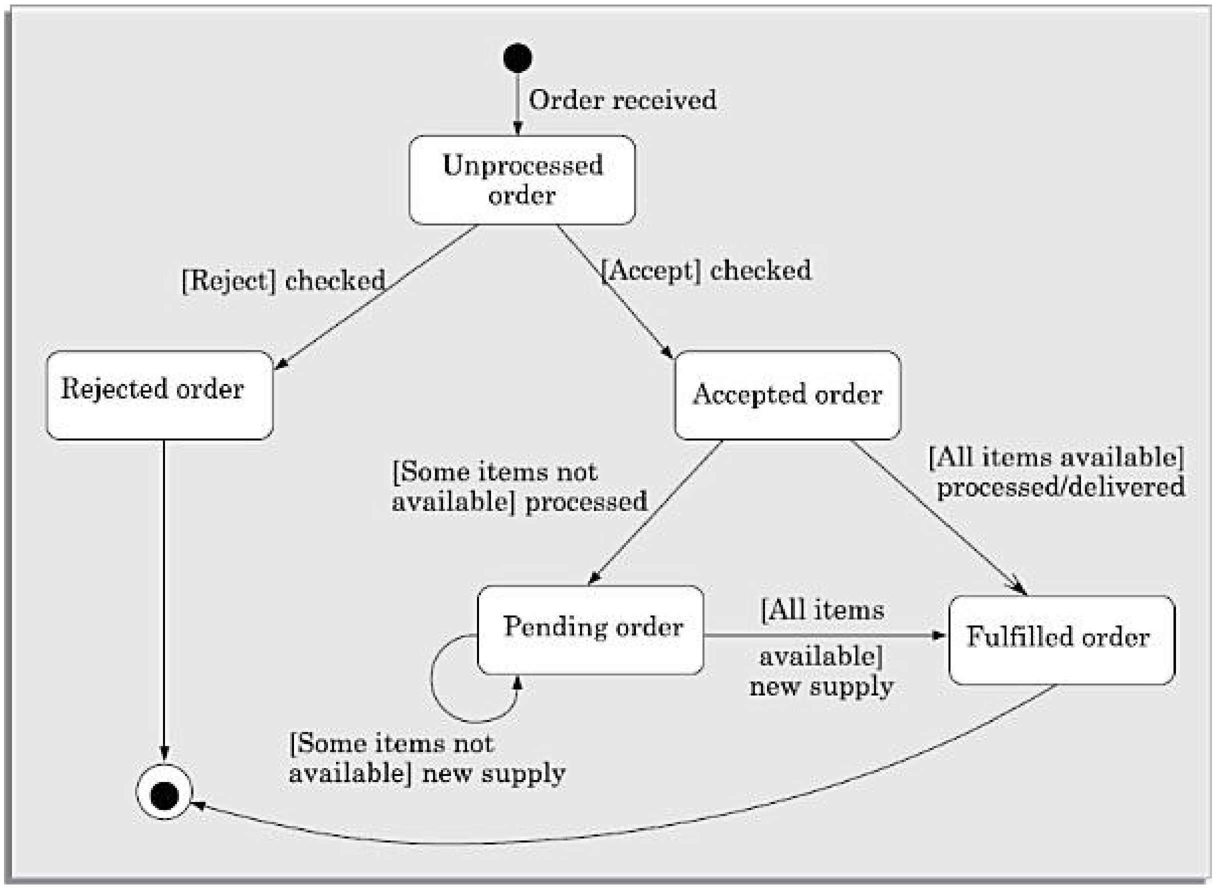

Order received
Unprocessed order
[Reject] checked
[Accept] checked
Rejected order
Accepted order
[Some items not available] processed
[All items available] processed/delivered
Pending order
[All items available] new supply
Fulfilled order
[Some items not available] new supply

Figure 7.33: State chart diagram for an order object.

In this text, we have kept our discussions on UML to a bare minimum and have concentrated only on those aspects that are necessary to solve moderate sized traditional software design problems.

Before concluding this chapter, we give an overview of some of the aspects that we had chosen to leave out. We first discuss the package and deployment diagrams. Since UML has undergone a significant change with the release of UML 2.0 in 2003. We briefly mention the highlights of the improvements brought about UML 2.0 over the UML 1.X which was our focus so far. This significant revision was necessitated to make UML applicable to the development of software for emerging embedded and telecommunication domains.

7.9.1 Package, Component, and Deployment Diagrams

In the following subsections we provide a brief overview of the package, component, and deployment diagrams:

Package diagram

A package is a grouping of several classes. In fact, a package diagram can be used to group any UML artifacts. We had already discussed packaging of use cases in Section 7.4.6. Packages are popular way of organising source code files. Java packages are a good example which can be modelled using a package diagram. Such package diagrams show the different class groups (packages) and their inter dependencies. These are very useful to document organisation of source files for large projects that have a large number of program files. An example of a package diagram has been shown in Figure 7.34.

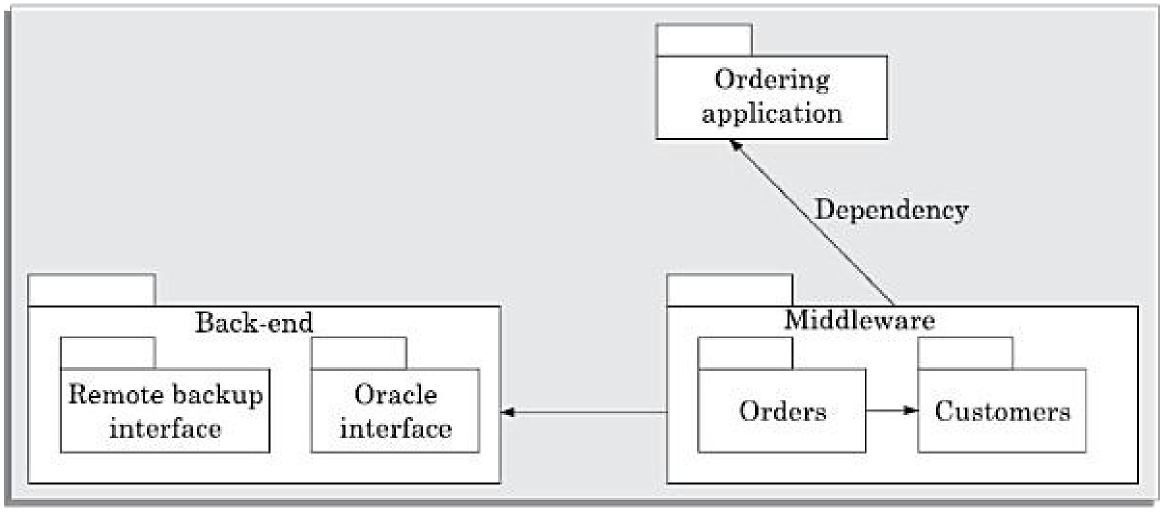


Figure 7.34: An example package diagram.

Note, that a package may contain further packages.

Component diagram

A component represents a piece of software that can be independently purchased, upgraded, and integrated into an existing software. A component diagram can be used to represent the physical structure of an implementation in terms of the various components of the system. A component diagram is typically used to achieve the following purposes:

- Organise source code to be able to construct executable releases.
- Specify dependencies among different components.

A package diagram can be used to provide a high-level view of each component in terms the different classes it contains.

Deployment diagram

The deployment diagram shows the environmental view of a system. That is, it captures the environment in which the software solution is implemented. In other words, a deployment diagram shows how a software system will be physically deployed in the hardware environment. That is, which component will execute on which hardware component and how they will they communicate with each other. Since the diagram models the run time architecture of an application, this diagram can be very useful to the system's operation staff.

The environmental view provided by the deployment diagram is important for complex and large software solutions that run on hardware systems comprising multiple components. In this case, deployment diagram provides an overview of how the different components are distributed among the different hardware components of the system.

7.9.2 UML 2.0

UML 1.X lacked a few specialised capabilities that made it difficult to use in some non- traditional domains. Some of the features that prominently lacked in UML 1.X include lack of support for representation of the following—concurrent execution of methods, development domain, asynchronous messages, events, ports, and active objects. In many applications, including the embedded and telecommunication software development, capability to model timing requirements using a timing diagram was urgently required to make UML applicable in these important segments of software development. Further, certain changes were required to support interoperability among UML-based CASE tools using XML metadata interchange (XMI).

UML 2.0 defines thirteen types of diagrams, divided into three categories as follows:

Structure diagrams: These include the class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram.

Behaviour diagrams: These diagrams include the use case diagram, activity diagram, and state machine diagram.

Interaction diagrams: These diagrams include the sequence diagram, communication diagram, timing diagram, and interaction overview diagram. The collaboration diagram of UML 1.X has been renamed in UML 2.0 as communication diagram. This renaming was necessary as the earlier name was somewhat misleading, it shows the communications among the classes during the execution of a use case rather than showing collaborative problem solving.

Chapter 10

CODING AND TESTING

In this chapter, we will discuss the coding and testing phases of the software life cycle.

Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.

In the coding phase, every module specified in the design document is coded and unit tested. During unit testing, each module is tested in isolation from other modules. That is, a module is tested independently as and when its coding is complete.

After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.

Integration and testing of modules is carried out according to an integration plan. The integration plan, according to which different modules are integrated together, usually envisages integration of modules through a number of steps. During each integration step, a number of modules are added to the partially integrated system and the resultant system is tested. The full product takes shape only after all the modules have been integrated together. System testing is conducted on the full product. During system testing, the product is tested against its requirements as recorded in the SRS document.

We had already pointed out in Chapter 2 that testing is an important phase in software development and typically requires the maximum effort among all the development phases. Usually, testing of a professional software is carried out using a large number of test cases. It is usually the case that many of the different test cases can be executed in parallel by different team members. Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed. In a typical development organisation, at any time, the maximum number of software engineers can be found to be engaged in testing activities. It is not very surprising then that in the software industry there is always a large demand for software test engineers. However, many novice engineers bear the wrong impression that testing is a secondary activity and that it is intellectually not as stimulating as the activities associated with the other development phases.

> Over the years, the general perception of testing as monkeys typing in random data and trying to crash the system has changed. Now testers are looked upon as masters of specialised concepts, techniques, and tools.

As we shall soon realize, testing a software product is as much challenging as initial development activities such as specifications, design, and coding. Moreover, testing involves a lot of creative thinking.

In this Chapter, we first discuss some important issues associated with the activities undertaken in the coding phase. Subsequently, we focus on various types of program testing techniques for procedural and object-oriented programs.

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?

It is mandatory for the programmers to follow the coding standards. Compliance of their code to coding standards is verified during code inspection. Any code that does not conform to the coding standards is rejected during code review and the code is reworked by the concerned programmer. In contrast, coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

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. Subsequently, we discuss code review techniques. We then discuss software documentation in Section 10.3.

10.1.1 Coding Standards and Guidelines

Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

Representative coding standards

Rules for limiting the use of globals: These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.

Standard headers for different modules: The header of different modules should have standard format and information for ease of understanding and maintenance. The following is an example of header format that is being used in some companies:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module. This is a small writeup about what the module does.
- Different functions supported in the module, along with their input/output parameters.
- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., GlobalData) and local variable names start with small letters (e.g., localData). Constant names should be formed using capital letters only (e.g., CONSTDATA).

Conventions regarding error return values and exception handling mechanisms: The way error conditions are reported by different functions in a program should be standard within an organisation. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.

Representative coding guidelines: The following are some representative coding guidelines that are recommended by many software development organisations. Wherever necessary, the rationale behind these guidelines is also mentioned.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.

Avoid obscure side effects: The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, suppose the value of a global variable is changed or some file I/O is performed obscurely in a called module. That is, this is difficult to infer from the function's name and header information. Then, it would be really hard to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers make use of a temporary loop variable for also computing and storing the final result. The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used. However, there are sev eral things wrong with this approach and hence should be avoided. Some of the problems caused by the use of a variable for multiple purposes are as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that cannot be a float type.

Code should be well-documented: As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

Length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use GO TO statements: Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

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 ty pes 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 main objective of code walkthrough is to discover the algorithmic and logical errors in 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 principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.

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?

Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand 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.

Example 10.1 Consider the following sentence: "The Gunning's fog index is based on the premise that use of short sentences and simple words makes a document easy to understand." Calculate its Fog index.

The fog index of the above example sentence is

$$0.4 \square (23/1) + (4/23) \square 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

10.4 TESTING

The aim of program testing is to help realiseidentify 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.

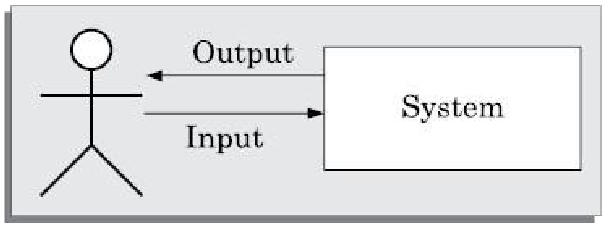


Figure 10.1: A simplified view of program testing.

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.

The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.

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.

Example 10.2 Can a designer's mistake give rise to a program error? Give an example of a designer's mistake and the corresponding program error.

Answer: Yes, a designer's mistake give rise to a program error. For example, a requirement might be overlooked by the designer, which can lead to it being overlooked in the code as well.

- A **failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution. An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:

– The result computed by a program is 0, when the correct result is 10.
– A program crashes on an input.
– A robot fails to avoid an obstacle and collides with it.

It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.

Example 10.3 Give an example of a program error that may not cause any failure.

Answer: Consider the following C program segment:

```
int markList[1:10]; /* mark list of 10 students*/
int roll;           /* student roll number*/
      ...
if(roll>0)
      markList[roll]=mark;
else
      markList[roll]=0;
```

In the above code, if the variable roll assumes zero or some negative value under some circumstances, then an array index out of bound type of error would result. However, it may be the case that for all allowed input values the variable roll is always assigned positive values. Then, the else clause is unreachable and no failure would occur. Thus, even if an error is present in the code, it does not show up as an error since it is unreachable for normal input values.

Explanation: An array index out of bound type of error is said to occur, when the array index variable assumes a value beyond the array bounds.

- A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode. As an example, consider the different execution modes of a certain text editor software. The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display. In simple words, we can say that a test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

A n example of a test case is—[input: "abc", state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string a b c would be displayed.

- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed. An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.
- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.
- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality. A test

case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system. As one example each of a positive test case and a negative test case, consider a program to manage user login. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the the login functionality validates and admits a user with wrong or bogus login user name or password.

- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

Example 10.4 Suppose two programs have been written to implement essentially the same functionality. How can you determine which of these is more testable?

Answer: A program is more testable, if it can be adequately tested with less number of test cases. Obviously, a less complex program is more testable. The complexity of a program can be measured using several types of metrics such as number of decision statements used in the program. Thus, a more testable program should have a lower structural complexity metric.

- A **failure mode** of a software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.

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.

We can therefore say that the primary objective of the verification steps are to determine whether the steps in product development are being carried out alright, whereas validation is carried out towards the end of the development process to determine whether the right product has been developed.

- Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle. The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors. Phase containment of errors can reduce the effort required for correcting bugs. For example, if a design problem is 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 system testing activities, thereby incurring higher cost.

While verification is concerned with phase containment of errors, the aim of validation is to check whether the deliverable software is error free.

We can consider the verification and validation techniques to be different types of bug filters. To achieve high product reliability in a cost-effective manner, a development team needs to perform both verification and validation activities. The activities involved in these two types of bug detection techniques together are called the "V and V" activities.

Based on the above discussions, we can conclude that:

Error detection techniques = Verification techniques + Validation techniques

Example 10.5 Is it at all possible to develop a highly reliable software, using validation techniques alone? If so, can we say that all verification techniques are redundant?

Answer: It is possible to develop a highly reliable software using validation techniques alone. However, this would cause the development cost to increase drastically. Verification techniques help achieve phase containment of errors and provide a means to cost-effectively remove bugs.

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.

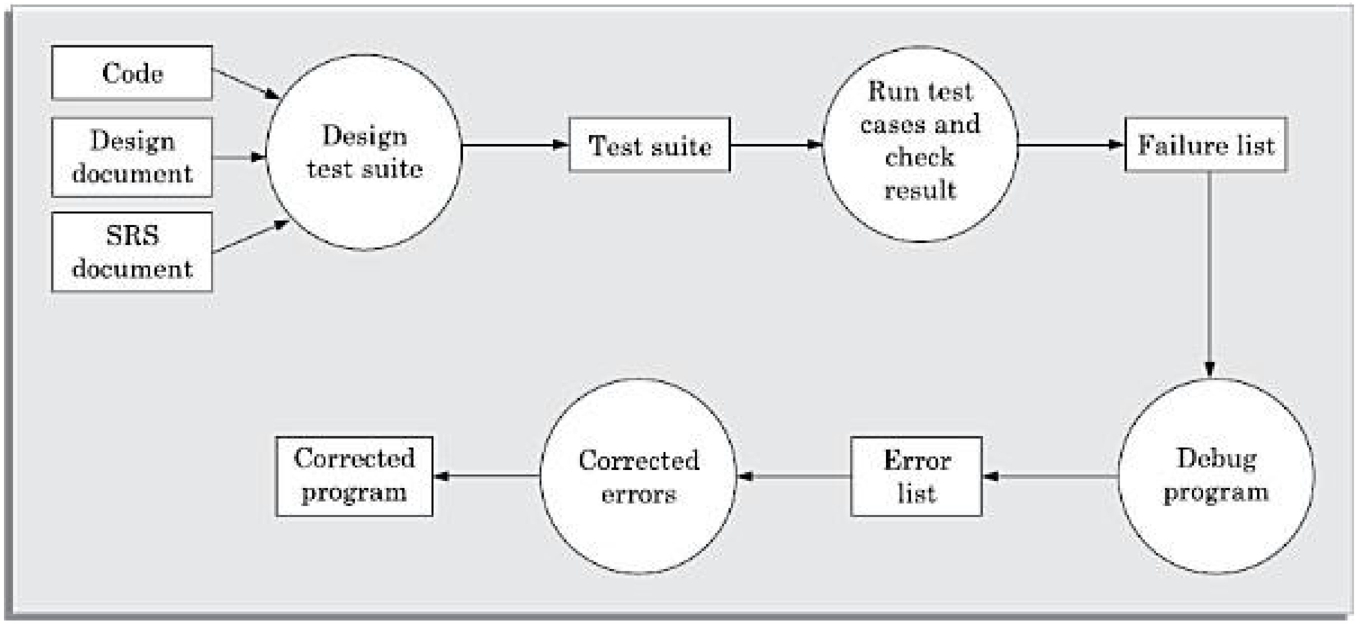


Figure 10.2: Testing process.

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:

When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite, That is, they do not help detect any additional defects not already being detected by other test cases in the suite.

Testing a software using a large collection of randomly selected test cases does not guarantee that all (or even most) of the errors in the system will be uncovered. Let us try to understand why the number of random test cases in a test suite, in general, does not indicate of the effectiveness of testing. Consider the following example code segment which determines the greater of two integer values x and y. This code segment has a simple programming error:

```
if (x>y) max = x;
else max = x;
```

For the given code segment, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error. All the test cases in the larger test suite help detect the same error, while the other error in the code remains undetected. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that for effective testing, the test suite should be carefully designed rather than picked randomly.

We have already pointed out that exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or countably infinite. Therefore, to satisfactorily test a software with minimum cost, we must design a minimal test suite that is of reasonable size and can uncover as many existing errors in the system as possible. To reduce testing cost and at the same time to make testing more effective, systematic approaches have been developed to design a small test suite that can detect most, if not all failures.

A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.

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.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.

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.

In the following sections, we discuss the different levels of testing. It should be borne in mind in all our subsequent discussions that unit testing is carried out in the coding phase itself as soon as coding of a module is complete. On the other hand, integration and system testing are carried out during the testing phase.

10.5 UNIT TESTING

Unit testing is undertaken after a module has been coded and reviewed. This activity is typically undertaken by the coder of the module himself in the coding phase. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed. In this section, we first discuss the environment needed to perform unit testing.

Driver and stub modules

In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:

- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub: The role of stub and driver modules is pictorially shown in Figure 10.3. A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

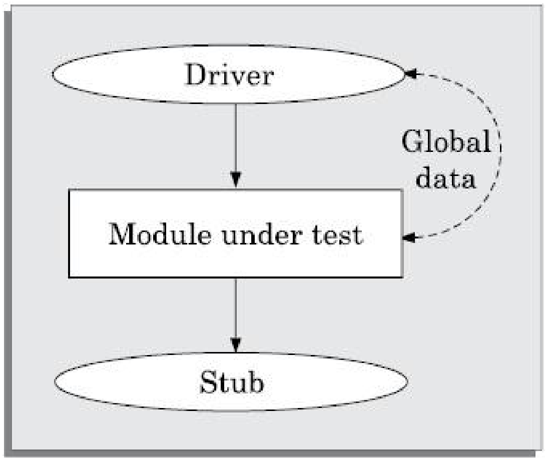


Figure 10.3: Unit testing with the help of driver and stub modules.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

10.6 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.

10.6.1 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.

The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

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.

Example 10.6 For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {–5,500,6000}.

Example 10.7 Design the equivalence class test cases for a program that reads two integer pairs (m_1, c_1) and (m_2, c_2) defining two straight lines of the form `y=mx+c`. The program computes the intersection point of the two straight lines and displays the point of intersection.

Answer: The equivalence classes are the following:

- Parallel lines ($m_1 = m_2$, c_1 □ c_2)
- Intersecting lines (m_1 □ m_2)
- Coincident lines ($m_1 = m_2$, $c_1 = c_2$)

Now, selecting one representative value from each equivalence class, we get the required equivalence class test suite {(2,2)(2,5),(5,5)(7,7), (10,10)

(10,10)}.

Example 10.8 Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

Answer: The equivalence classes are the leaf level classes shown in Figure 10.4. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.

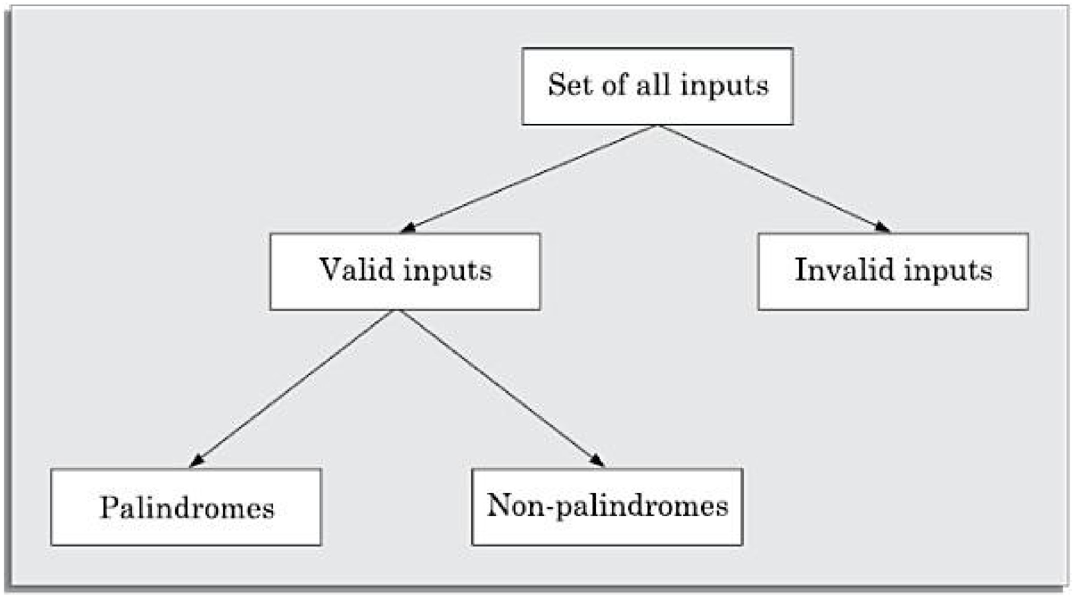


Figure 10.4: Equivalence classes for Example 10.6.

10.6.2 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}.

Example 10.9 For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

Example 10.10 Design boundary value test suite for the function described in Example 10.6.

Answer: The equivalence classes have been showed in Figure 10.5. There is a boundary between the valid and invalid equivalence classes. Thus, the boundary value test suite is {abcdefg, abcdef}.

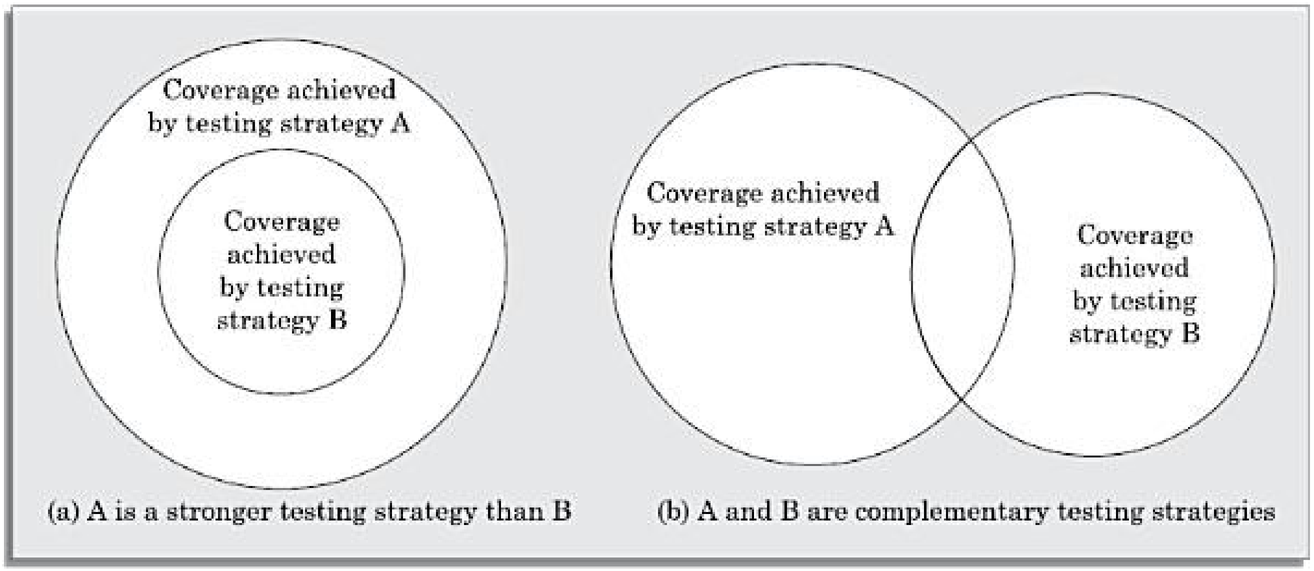

Coverage achieved by testing strategy A
Coverage achieved by testing strategy B
(a) A is a stronger testing strategy than B
Coverage achieved by testing strategy A
Coverage achieved by testing strategy B
(b) A and B are complementary testing strategies

Figure 10.5: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

10.6.3 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.

10.7 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.

10.7.1 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.

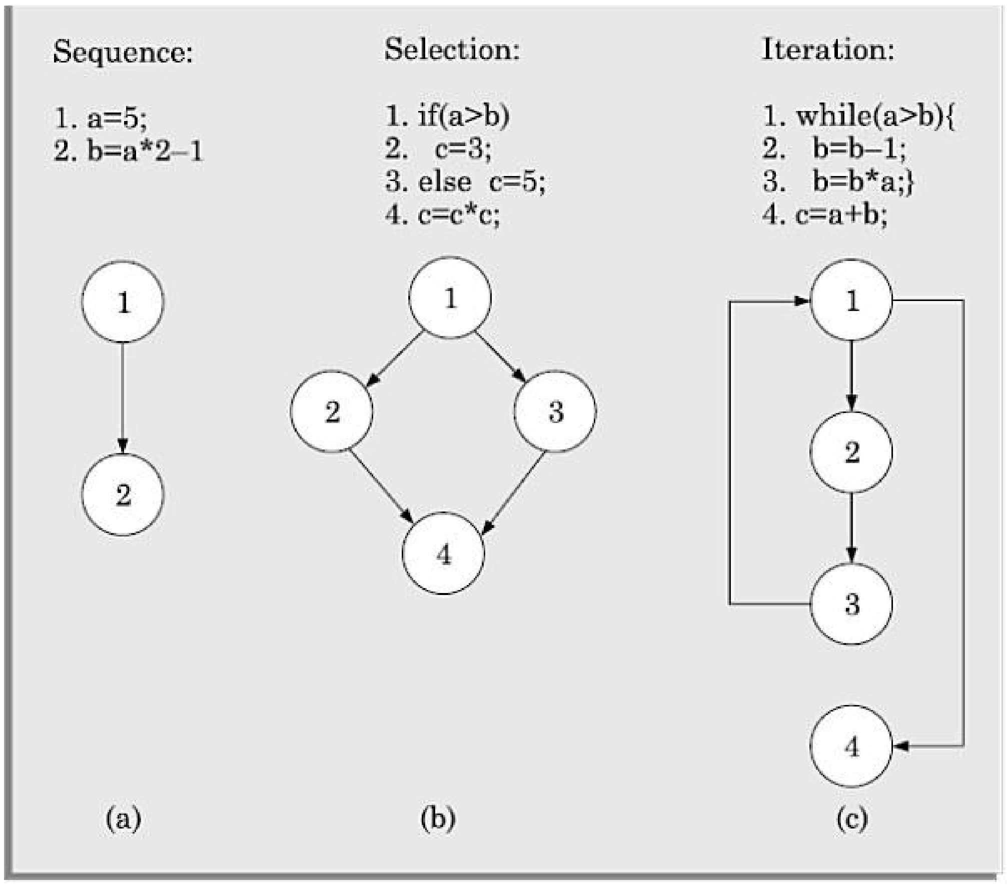


Figure 10.6: Illustration of stronger, weaker, and complementary testing strategies.

A test suite should, however, be enriched by using various complementary testing strategies.

We need to point out that coverage-based testing is frequently used to check the quality of testing achieved by a test suite. It is hard to manually design a test suite to achieve a specific coverage for a non-trivial program.

10.7.2 Statement Coverage

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.

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.

Example 10.11 Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
   int x,y;
{
    while (x != y){
    if (x>y) then
    x=x-y;
    else y=y-x;
    }
    return x;
}
```

Answer: To design the test cases for the statement coverage, the conditional expression of the `while` statement needs to be made true and the conditional expression of the `if` statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, all statements of the program would be executed at least once.

10.7.3 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.

Example 10.12 For the program of Example 10.11, determine a test suite to achieve branch coverage.

Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage.

It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Theorem 10.1 Branch coverage-based testing is stronger than statement coverage-based testing.

Proof: We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.

(a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).

(b) To show that statement coverage does not ensure branch coverage, it would be sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

```
if(x>2) x+=1;
```

The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition (x > 2) is not made false by any test case in the suite.

10.7.4 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 ((c_1 .and.c_2).or.c_3). A test suite would achieve MC coverage, if all the component conditions c_1, c_2 and c_3 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.

Example 10.13 Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.

Answer: Consider the following C program segment:

```
if(temperature>150 || temperature>50)
 setWarningLightOn();
```

The program segment has a bug in the second component condition, it should have been `temperature<50`. The test suite {temperature=160, temperature=40} achieves branch coverage. But, it is not able to check that `setWarningLightOn();` should not be called for temperature values within 150 and 50.

10.7.5 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)

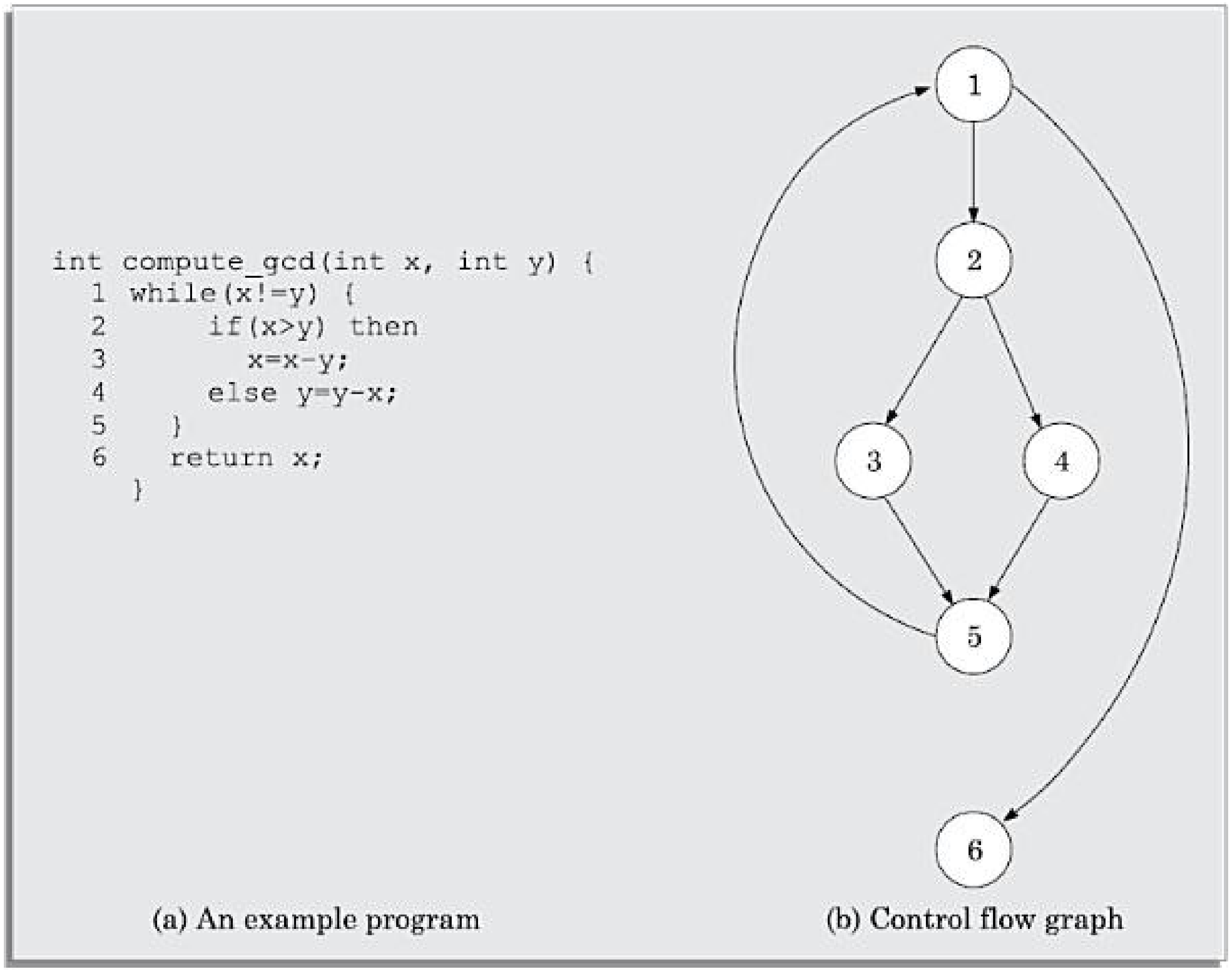
A control flow graph describes how the control flows through the program. We can define a control flow graph as the following:

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.

We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only. Figure 10.5 summarises how the CFG for these three types of constructs can be drawn. The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop (iteration) construct can be drawn. For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop. Using these basic ideas, the CFG of the program given in Figure 10.7(a) can be drawn as shown in Figure 10.7(b).



```
int compute_gcd(int x, int y) {
  1  while(x!=y) {
  2      if(x>y) then
  3         x=x-y;
  4      else y=y-x;
  5    }
  6    return x;
     }
```

Figure 10.7: Control flow diagram of an example program.

Path

A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal nodes when it contains multiple exit or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops. For example, in Figure 10.5(c), there can be an infinite number of paths such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (or basis paths). Let us now discuss what are linearly independent paths and how to determine these in a program.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because, any path having a new node would automatically have a new edge. An alternative definition of a linearly independent set of paths [McCabe76] is the following:

If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.

According to the above definition of a linearly independent set of paths, for any path in the set, its subpath cannot be a member of the set. In fact, any arbitrary path of a program, can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the "basis" for all the paths of a program. Please note that there may not always exist a unique basis set for a program and several basis sets for the same program can usually be determined.

Even though it is straight forward to identify the linearly independent paths for simple programs, for more complex programs it is not easy to determine the number of independent paths. In this context, McCabe's cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Though the McCabe's metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.

10.7.6 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.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E - N + 2$$

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 10.7, E = 7 and N = 6. Therefore, the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows —In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:

```
V(G) = Total number of non-overlapping bounded areas + 1
```

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar (i.e., how ever you draw the graph, two or more edges always intersect). Actually, it can be shown that control flow representation of structured programs always yields planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity does not apply.

The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program. Consider the CFG example shown in Figure 10.7. From a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computed with this method is also 2+1=3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the method for computing CFGs can easily be automated. That is, the McCabe's metric computations methods 1 and 3 can be easily coded into a program that can be used to automatically determine the cyclomatic complexities of arbitrary programs.

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe's metric is equal to N + 1.

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.

Uses of McCabe's cyclomatic complexity metric

Beside its use in path testing, cyclomatic complexity of programs has many other interesting applications such as the following:

Estimation of structural complexity of code: McCabe's cyclomatic complexity is a measure of the structural complexity of a program. The reason for this is that it is computed based on the code structure (number of decision and iteration constructs used). Intuitively, the McCabe's complexity metric correlates with the difficulty level of understanding a program, since one understands a program by understanding the computations carried out along all independent paths of the program.

Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program.

In view of the above result, from the maintenance perspective, it makes good sense to limit the cyclomatic complexity of the different functions to some reasonable value. Good software development organisations usually restrict the cyclomatic complexity of different functions to a maximum value of ten or so. This is in contrast to the computational complexity that is based on the execution of the program statements.

Estimation of testing effort: Cyclomatic complexity is a measure of the maximum number of basis paths. Thus, it indicates the minimum number of test cases required to achieve path coverage. Therefore, the testing effort and the time required to test a piece of code satisfactorily is proportional to the cyclomatic complexity of the code. To reduce testing effort, it is necessary to restrict the cyclomatic complexity of every function to seven.

Estimation of program reliability: Experimental studies indicate there exists a clear relationship between the McCabe's metric and the number of errors latent in the code after testing. This relationship exists possibly due to the correlation of cyclomatic complexity with the structural complexity of code. Usually the larger is the structural complexity, the more difficult it is to test and debug the code.

10.7.7 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.

The definition-use chain (or DU chain) of a variable X is of the form [X, S, S1], where S and S1 are statement numbers, such that X □ DEF(S) and X □ USES(S1), and the definition of X in the statement S is live at statement S1 . One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are especially useful for testing programs containing nested if and loop statements.

10.7.8 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.

A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant. However, it is not this straightforward. Remember that there is a possibility of a mutated program to be an equivalent program. When this is the case, it is futile to try to design a test case that would identify the error.

An important advantage of mutation testing is that it can be automated to a great extent. The process of generation of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be simple program alterations such as—deleting a statement, deleting a variable definition, changing the type of an arithmetic operator (e.g., + to -), changing a logical operator (`and` to `or`) changing the value of a constant, changing the data type of a variable, etc. A major pitfall of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Mutation testing involves generating a large number of mutants. Also each mutant needs to be tested with the full test suite. Obviously therefore, mutation testing is not suitable for manual testing. Mutation testing is most suitable to be used in conjunction of some testing tool that should automatically generate the mutants and run the test suite automatically on each mutant. At present, several test tools are available that automatically generate mutants for a given program.

10.8 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 and disadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

10.8.1 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.

10.8.2 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.

10.9 PROGRAM ANALYSIS TOOLS

A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc. We can classify various program analysis tools into the following two broad categories:

- Static analysis tools
- Dynamic analysis tools

These two categories of program analysis tools are discussed in the following subsection.

10.9.1 Static Analysis Tools

Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyse the source code to compute certain metrics characterising the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:

- To what extent the coding standards have been adhered to?
- Whether certain programming errors such as uninitialised variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist? A list of all such errors is displayed.

Code review techniques such as code walkthrough and code inspection discussed in Sections 10.2.1 and 10.2.2 can be considered as static analysis methods since those target to detect errors based on analysing the source code. However, strictly speaking, this is not true since we are using the term static program analysis to denote automated analysis tools. On the other hand, a compiler can be considered to be a type of a static program analysis tool.

A major practical limitation of the static analysis tools lies in their inability to analyse run-time information such as dynamic memory references using pointer variables and pointer arithmetic, etc. In a high level programming languages, pointer variables and dynamic memory allocation provide the capability for dynamic memory references. However, dynamic memory referencing is a major source of programming errors in a program.

Static analysis tools often summarise the results of analysis of every function in a polar chart known as Kiviat Chart. A Kiviat Chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead's metrics, etc.

10.9.2 Dynamic Analysis Tools

Dynamic program analysis tools can be used to evaluate several program characteristics based on an analysis of the run time behaviour of a program. These tools usually record and analyse the actual behaviour of a program while it is being executed. A dynamic program analysis tool (also called a dynamic analyser) usually collects execution trace information by instrumenting the code. Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.

An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.

After a software has been tested with its full test suite and its behaviour recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program. For example, the dynamic analysis tool can report the statement, branch, and path coverage achieved by a test suite. If the coverage achieved is not satisfactory more test cases can be designed, added to the test suite, and run. Further, dynamic analysis results can help eliminate redundant test cases from a test suite.

Normally the dynamic analysis results are reported in the form of a histogram or pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily to provide evidence that thorough testing has been carried out.

10.10 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.

The objective of integration testing is to check whether the different modules of a program interface with each other properly.

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.

10.10.1 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.

10.11 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 are present 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.

10.11.1 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.

10.11.2 Do Various Ob ject-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.

Even if the base class class has been thoroughly tested, the methods inherited from the base class need to be tested again in the derived class.

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.

10.11.3 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.

10.11.4 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.

For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs. These are called the grey-box test cases.

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.

10.11.5 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.

10.12 SYSTEM TESTING

After all the units of a program have been integrated together and tested, system testing is taken up.

System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or object-oriented) is immaterial.

There are essentially three main kinds of system testing depending on who carries out testing:

1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organisation.
2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
3. **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

In each of the above types of system tests, the test cases can be the same, but the difference is with respect to who designs test cases and carries out testing.

The system test cases can be classified into functionality and performance test cases.

Before a fully integrated system is accepted for system testing, smoke testing is performed. Smoke testing is done to check whether at least the main functionalities of the software are working properly. Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.

The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system. We have already discussed how to design the functionality test cases by using a black-box approach (in Section 10.5 in the context of unit testing). So, in the following subsection we discuss only smoke and performance testing.

10.12.1 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.

10.12.2 Performance Testing

Performance testing is an important type of system testing.

Performance testing is carried out to check whether the system meets the non-functional requirements identified in the SRS document.

There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document. All performance tests can be considered as black-box tests.

Stress testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilisation of memory, etc., are tested beyond the designed capacity. For example, suppose an operating system is supposed to support fifteen concurrent transactions, then the system is stressed by attempting to initiate fifteen or more transactions simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours. For example, if the corresponding non-functional requirement states that the response time should not be more than twenty secs per transaction when sixty concurrent users are working, then during stress testing the response time is checked with exactly sixty users working simultaneously.

Volume testing

Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations. For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Configuration testing

Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements. Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing. The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

Compatibility testing

This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.). Compatibility aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression testing

This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc. Regression testing is also discussed in Section 10.13.

Recovery testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance testing

This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation testing

It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.

Usability testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough. Therefore, the GUI has to be checked against the checklist we discussed in Sec. 9.5.6.

Security testing

Security testing is essential for software that handle or process confidential data that is to be gurarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

10.12.3 Error Seeding

Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in a software.

Error seeding, as the name implies, it involves seeding the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

- The number of errors remaining in the product.
- The effectiveness of the testing strategy.

Let N be the total number of defects in the system, and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing. Therefore, we get:

$$\frac{n}{N} = \frac{s}{S}$$

or

$$N = S \times \frac{n}{s}$$

Defects still remaining in the program after testing can be given by:

$$N - n = n \times \frac{(S-1)}{s}$$

Error seeding works satisfactorily only if the kind seeded errors and their frequency of occurrence matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that are latent and their frequency of occurrence can be estimated by analyzing historical data collected from similar projects. That is, the data collected is regarding the types and the frequency of latent errors for all earlier related projects. This gives an indication of the types (and the frequency) of errors that are likely to have been committed in the program under consideration. Based on these data, the different types of errors with the required frequency of occurrence can be seeded.

10.13 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.

SUMMARY

UNIT-5

SOFTWARE RELIABILITY

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently is the corresponding instruction executed.

Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the "correctly" implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

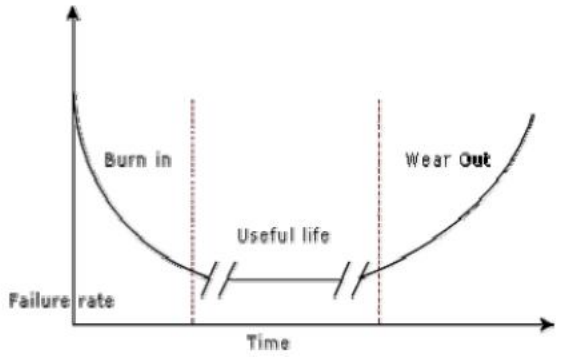
The reasons why software reliability is difficult to measure can be summarized as follows:

- 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 highly observer dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

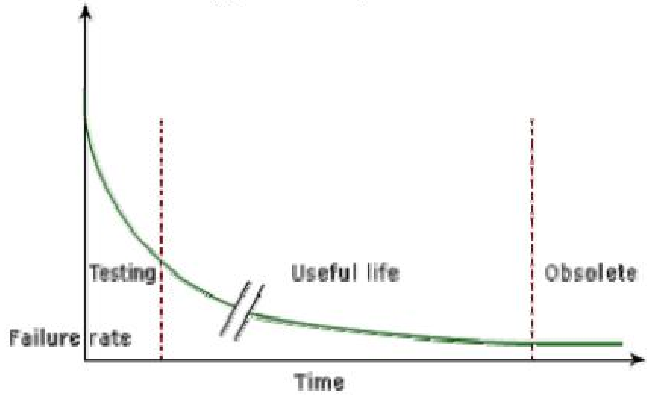
Hardware reliability vs. software reliability differ

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase).

The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 5.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, 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 error corrections occurs and the failure rate remains unchanged.



(a) Hardware product



(b) Software product

Fig 5.1: Change in failure rate of a product

Reliability metrics

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 SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

• **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

• **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures

• **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 (MTBR). MTTF and MTTR can be combined to get the MTBR 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, 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. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

• **Availability.** Availability of a system is a measure of how likely shall 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, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

Classification of software failures

A possible classification of failures of software products into five different types is as follows:

• **Transient.** Transient failures occur only for certain input values while invoking a function of the system.

• **Permanent.** Permanent failures occur for all input values while invoking a function of the system.

• **Recoverable.** When recoverable failures occur, the system recovers with or without operator intervention.

• **Unrecoverable**. In unrecoverable failures, the system may need to be restarted.

• **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

Reliability growth models

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. 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 fig. 5.2. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.

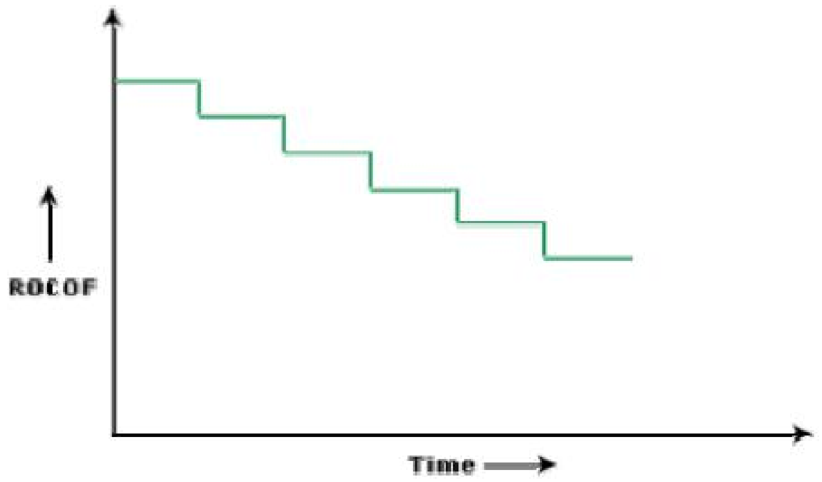


Fig. 5.2: Step function model of reliability growth

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 in reliability per repair decreases (Fig. 5.3). It treat's 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.

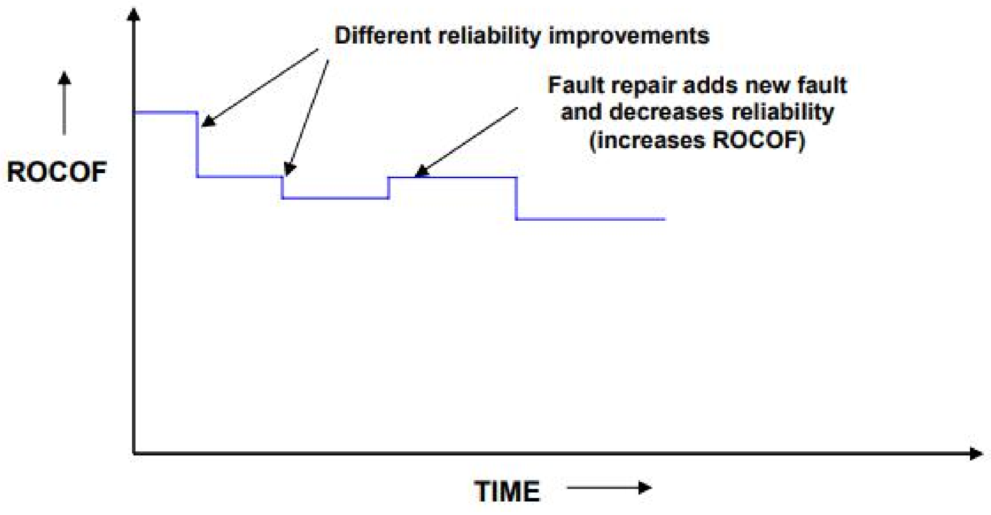


Fig. 5.3: Random-step function model of reliability growth

STATISTICAL TESTING

Statistical testing is a testing process whose objective is to determine the reliability of software products rather than discovering errors. Test cases are designed for statistical testing with an entirely different objective than those of conventional testing.

Operation profile
Different categories of users may use a software for different purposes. For example, a Librarian might use the library automation software to create member records, add books to the library, etc. whereas a library member might use to software to query about the availability of the book, or to issue and return books. Formally, the operation profile of software can be defined as the probability distribution of the input of an average user. If the input to a number of classes {Ci} is divided, the probability value of a class represent the probability of an average user selecting his next input from this class. Thus, the operation profile assigns a probability value Pi to each input class Ci.
Steps in statistical testing
Statistical testing allows one to concentrate on testing those parts of the system that are most likely to be used. The first step of statistical testing is to determine the operation profile of the software. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.
Advantages and disadvantages of statistical testing
Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users to be more reliable (than actually it is!). Reliability estimation using statistical testing is more accurate compared to those of other methods such as ROCOF, POFOD etc. But it is not easy to perform statistical testing properly. There is no simple and repeatable way of defining operation profiles. Also it is very much cumbersome to generate test cases for statistical testing cause the number of test cases with which the system is to be tested should be statistically significant.

SOFTWARE QUALITY
Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually 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. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.
The modern view of a quality associates with a software product several quality factors such as the following:
• **Portability**: A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.
• **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.
Software quality management system

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:
• **Managerial Structure and Individual Responsibilities.** A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have 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
- review of the quality system
- development of standards, procedures, and guidelines, etc.
- production of reports for the top management summarizing the effectiveness of the quality system in the organization.

SOFTWARE QUALITY MANAGEMENT SYSTEM

Quality systems have rapidly evolved over the last five 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 organizations have undergone through four stages of evolution as shown in the fig. 5.4. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization’s processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements.
TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance (as shown in fig. 5.4).

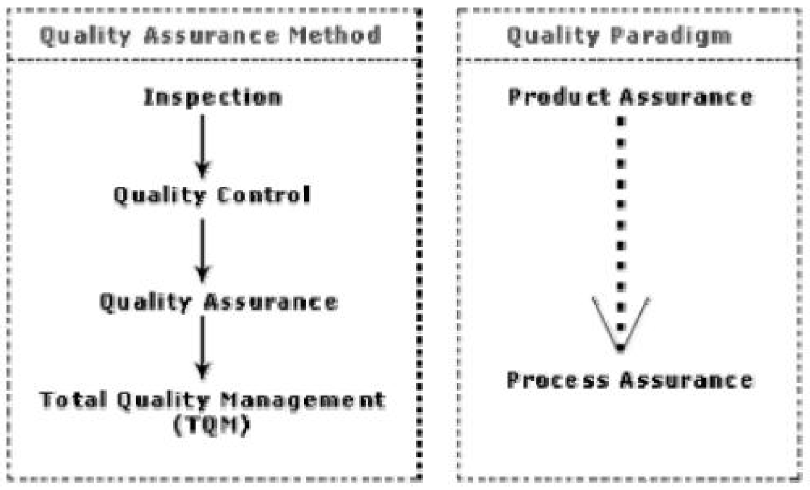


Fig. 5.4: Evolution of quality system and corresponding shift in the quality paradigm

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.

What is ISO 9000 Certification?

ISO 9000 certification serves as a reference for contract between independent parties. ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product it self.

The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

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 organizations 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.
ISO 9003: This standard applies to organisations involved only in installation and testing of products.

ISO 9000 for Software Industry

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:

1. 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.
2. 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.

Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry. Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry.

Why Get ISO 9000 Certification?

There is a mad scramble among software development organisations for obtaining ISO certification due to the benefits it offers. The following are the benefits:

1. Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification. This is especially true in the international market.
2. 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.
3. ISO 9000 makes the development process focused, efficient, and cost effective.
4. ISO 9000 certification points out the weak points of an organization and recommends remedial action.
5. ISO 9000 sets the basic framework for the development of an optimal process and TQM.

How to Get ISO 9000 Certification?

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.

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

- 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

- A quality system must be maintained and documented.

Contract reviews

- 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

- 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

- 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

- Purchased material, including bought-in software must be checked for conforming to requirements.

Purchaser supplied product

- Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product identification

- The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process control

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and testing

- 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

- If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

Inspection and test status

- The status of an item must be identified. In software terms this implies configuration management and release control.

Control of non-conforming product

- 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

- 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

- This clause deals with the storage, packing, and delivery of the software product.

Quality records

- 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

- Audits of the quality system must be carried out to ensure that it is effective.

Training

- Training needs must be identified and met.

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 organizational aspects should be addressed e.g., management reporting of the quality team.

5.5.7 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 measurable objectives for various roles and levels of the organisation. The new standard recognises that there can be many processes in an organisation.

Shortcomings of ISO 9000 Certification

Even though ISO 9000 is widely being used for setting up an effective

Quality system in an organisation, it suffers from several shortcomings. Some of these shortcoming of the ISO 9000 certification process are the following:

- ISO 9000 requires a software production process to be adhered to, but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool-proof and no international accredition agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accredition agencies and also among the registrars.
- Organisations getting ISO 9000 certification often tend to downplay domain expertise and the ingenuity of the developers. These organisations start to believe that since a good process is in place, the development results are truly person-independent. That is, any developer is as effective as any other developer in performing any particular software development activity. In manufacturing industry there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. Many areas of software development are so specialised that special expertise and experience in these areas (domain expertise) is required. Also, unlike in case of general product manufacturing, ingenuity and effectiveness of personal practices play an important part in determining the results produced by a developer. In other words, software development is a creative process and individual skills and experience are important.
- ISO 9000 does not automatically lead to continuous process improvement. In other words, it does not automatically lead to TQM.

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 organization 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. Capability evaluation is administered by

the contract awarding authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work. 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 depends 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. Configuration management practices are used for all project deliverables. Please remember that opportunity to repeat a process exists only when a company produces a family of products. Since the products are very similar, the success story on development of one product can repeated for another. In a nonrepeatable software development organisation, a software product

development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. On the other hand, in a non-repeatable software development organisation, the chances of successful completion of a software project is to a great extent depends on who the team members are. For this reason, the successful development of one product by such an organisation does not automatically imply that the next product development will be successful.

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. ISO 9000 aims at achieving this level.

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. For example, if from an analysis of the process measurement results, it is found that the code reviews are not very effective and a large number of errors are detected only during the unit testing, then the process would be fine tuned

to make the review more effective. Also, the lessons learned from specific projects are incorporated into the process. Continuous process improvement is achieved both by carefully analysing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. 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 organisationwide.
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.
Except for level 1, each maturity level is characterised by several key process areas (KPAs) that indicate the areas an organisation should focus to improve its software process to this level from the previous level. Each of the focus areas identifies a number of key practices or activities that need to be implemented. In other words, KPAs capture the focus areas of a level. The focus of each level and the corresponding key process areas are shown in the Table 11.1:

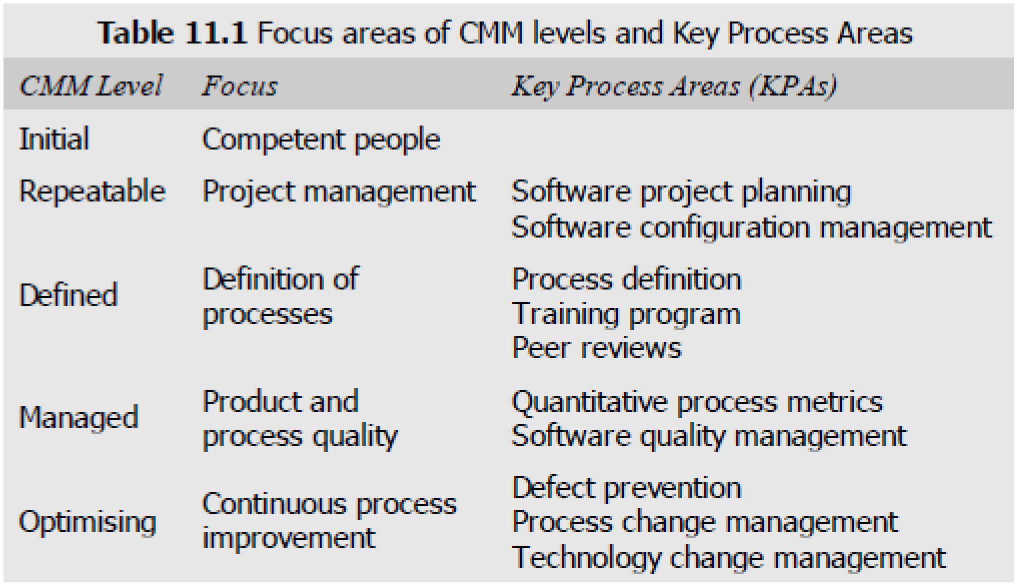


Table 11.1 Focus areas of CMM levels and Key Process Areas

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. For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.
Substantial evidence has now been accumulated which indicate that adopting SEI CMM has several business benefits. However, the organizations 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 the minimum 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.

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 organization 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.

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 organizations 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.

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. In 2002, CMMI Version 1.1 was released. Version 1.2 followed in 2006. 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. For example, the word “software” does not appear in definitions of CMMI. This unification of various types of domains into a single model makes CMMI extremely abstract. The CMMI, like its predecessor, describes five distinct levels of maturity.

Software Process Improvement and Capability Determination (SPICE)

SPICE stands for Software Process Improvement and Capability determination. It is an ISO standard (IEC 15504). It distinguishes different kinds of processes—engineering process, management process, customer-supplier, support. For each process, it defines six capability maturity levels. It integrates existing standards to provide a single process reference model and process assessment model that addresses broad categories of enterprise processes.

Personal Software Process (PSP)

PSP is based on the work of David Humphrey [Hum97]. PSP is a scaled down version of industrial software process discussed in the last section. PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyse, design, code, test, or document software products, but assumes that engineers use effective personal practices. PSP recognizes that the process for individual use is different from that necessary for a team.

The quality and productivity of an engineer is to a great extent dependent
on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating, planning, and tracking performance against plans, and provides a defined process which can be tuned by individuals.

Time measurement: PSP advocates that developers should rack the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-watch to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break, etc. An engineer should measure the time he spends for various development activities such as designing, writing code, testing, etc.

PSP Planning: Individuals must plan their project. Unless an individual properly plans his activities, disproportionately high effort may be spent on trivial activities and important activities may be compromised, leading to poor quality results. The developers must estimate the maximum, minimum, and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in Figure 11.4. While carrying out the different phases, an individual must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve his process, etc.

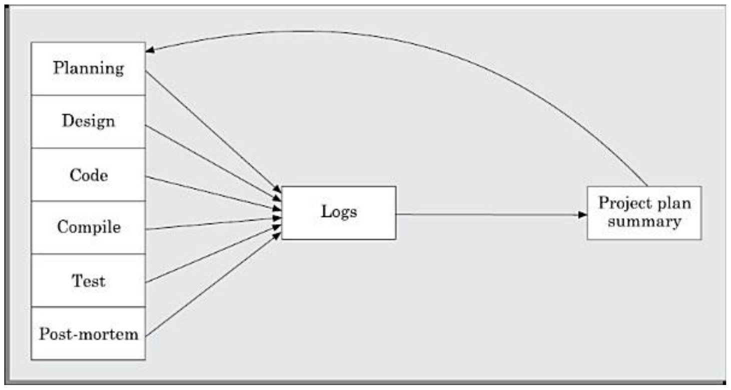


Figure 11.4: A schematic representation of PSP.

The PSP levels are summarised in Figure 11.5. PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed by analysing the defect data gathered from earlier projects.

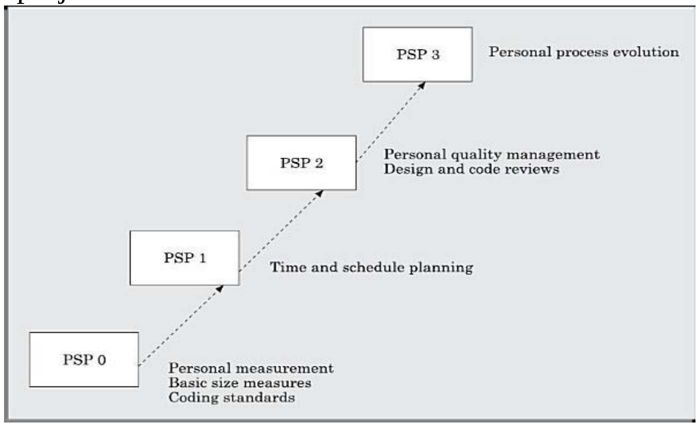


Figure 11.5: Levels of PSP.

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 impleme
nting Six Sigma quality, based on the similar change management philosophies and applications of tools.

SOFTWARE REUSE

Software products are expensive. Therefore, software project managers are always worried about the high cost of software development and are desperately looking for ways to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality. A reuse approach that is of late gaining prominence is component-based development. Component-based software development is different from the traditional software development in the sense that software is developed by assembling software from off-the-shelf components.
Software development with reuse is very similar to a modern hardware engineer building an electronic circuit by using standard types of ICs and other components. In this Chapter, we will review the state of art in software reuse.

what can be reused

Before discussing the details of reuse techniques, it is important to deliberate about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:
Requirements specification
Design
Code
Test cases
Knowledge
Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge is that a developer experienced in one type of product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

Issues

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.

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.

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.

For example, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. We can see that the domain analysis generalises the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalised.

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: H e r e , 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*.

14.4.2 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.

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.

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.

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 significant advantages over simple parameterised programs. The biggest of these is that the application generators can express the variant information in an appropriate language rather than being restricted to function parameters, named constants, or tables. The other advantages include fewer errors, easier to maintain, substantially reduced development effort, and the fact that one need not bother about the implementation details. Application generators are handicapped when it is necessary to support some new concepts or features.

Some application generators overcome this handicap through an escape mechanism. Programmers can write code in some 3GL through this mechanism.

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.

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.

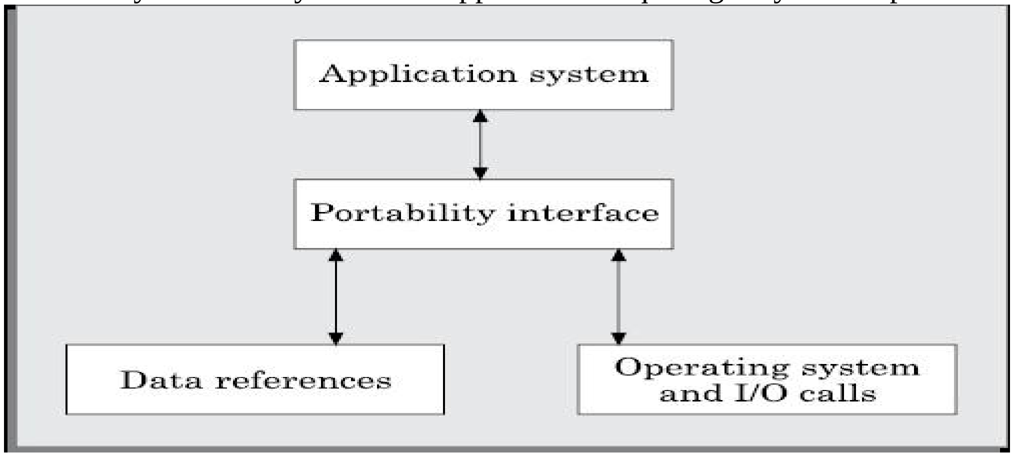


Figure 14.1: Improving reusability of a component by using a portability interface.

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.

Emerging trends:

CLIENT-SERVER SOFTWARE

In a client-server software, both clients and servers are essentially software components. A client is a consumer of services and a server is a provider of services. The client-server concept is not a new concept. It existed in the society since long. For example, a teacher may be a client of a doctor, and the doctor may in turn be a client of a barber, who in turn may be a client of the lawyer, and so forth. From this, we can observe that a server in some context can be a client in some other context. So, clients and servers can be considered to be mere roles.

Considering the level of popularity of the client-server paradigm in the context of software development, there must be several advantages accruing from adopting this concept. Let us deliberate on the important advantages of the client-server paradigm.

Advantages of client-server software

There are many reasons for the popularity of client-server software. A few important reasons are as follows:

Concurrency: A client-server software divides the computing work among many different client and server components that could be residing on different machines. Thus client-server solutions are inherently concurrent and as a result offer the advantage of faster processing.

Loose coupling: Client and server components are inherently looselycoupled, making these easy to understand and develop.

Flexibility: A client-server software i s flexible in the sense that clients and servers can be attached and removed as and when required. Also, clients can access the servers from anywhere.

Cost-effectiveness: The client-server paradigm usually leads to cost effective solutions. Clients usually run on cheap desktop computers, whereas severs may run on sophisticated and expensive computers. Even to use a sophisticated software, one needs to own only a cheap client machine to invoke the server.

Heterogeneous hardware: In a client-server solution, it is easy to have specialised servers that can efficiently solve specific problems. It is possible to efficiently integrate heterogeneous computing platforms to support the requirements of different types of server software.

Fault-tolerance: Client-server solutions are usually fault-tolerant. It is possible to have many servers providing the same service. If one server becomes unavailable, then client requests can be directed to any other working server.

Mobile computing: Mobile computing implicitly requires uses of clientserver technique. Cell phones are, of late, evolving as handheld computing and communicating devices and are being provided with small processing power, keyboard, small memory, and LCD display. The handhelds have limited processing power and storage capacity, and therefore can act only as clients. To perform any non-trivial task, the handheld computers can possibly only support the necessary user interface to place requests on some remote servers.

Application service provisioning: There are many application software products that are extremely expensive to own. A client-server based approach can be used to make these software products affordable for use. In this approach, a n application service provider (ASP) would own it, and the users would pay the ASP based on the charges per unit time of usage.

Component-based development: Client-server paradigm fits well with the component- based software development. Component-based software development holds out the promise of achieving substantial reductions to cost and delivery time and at the same time achieve increased product reliability.

Component-based development is similar to the way hardware equipments are being constructed cost-effectively. A hardware developer achieves cost, effort, and time savings in an equipment development by integrating pre-built components (ICs) purchased off-the-shelf on a printed circuit board (PCB).

As discussed, advantages of the client-server software paradigm are numerous. No wonder that the client-server paradigm has become extremely popular. However, before we discuss more details of this technology, it is important to know the important shortcomings of it as well.

Disadvantages of client-server software

There are several disadvantages of client-server software development. The main disadvantages are:

Security: In a monolithic application, addressing the security concerns is much easier as compared to client-server implementations. A client-server based software provides many flexibilities. For example, a client can connect to a server from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security of a client-server system is a very challenging task.

Servers can be bottlenecks: Servers can turn out to be bottlenecks because many clients might try to connect to a server at the same time. This problem arises due to the flexibility given that any client can connect anytime required.

Compatibility: Clients and servers may not be compatible to each other. Since the client and server components may be manufactured by different vendors, they may not be compatible with respect to data types, languages, number representation, etc.

Inconsistency: Replication of servers can potentially create problems as whenever there is replication of data, there is a danger of the data becoming inconsistent.

CLIENT-SERVER ARCHITECTURES

The simplest way to connect clients and servers is by using a two-tier architecture shown in Figure 15.1(a). In a two-tier architecture, any client can get service from any server by sending a request over the network.

Limitations of two-tier client-server architecture

A two-tier architecture for client-server applications though is an intuitively obvious solution, but it turns out to be not practically usable. The main problem is that client and server components are usually manufactured by different vendors, who may adopt their own interfacing and implementation solutions. As a result, the different components may not interface with (talk to) each other easily.

Three-tier client-server architecture

The three-tier architecture overcomes the main limitations of the two tier architecture. In the three-tier architecture, a middleware is added between client and the server components as shown in Figure 15.1(b).
The middleware keeps track of all servers. It also translates client requests into server understandable form. For example, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer to the client.

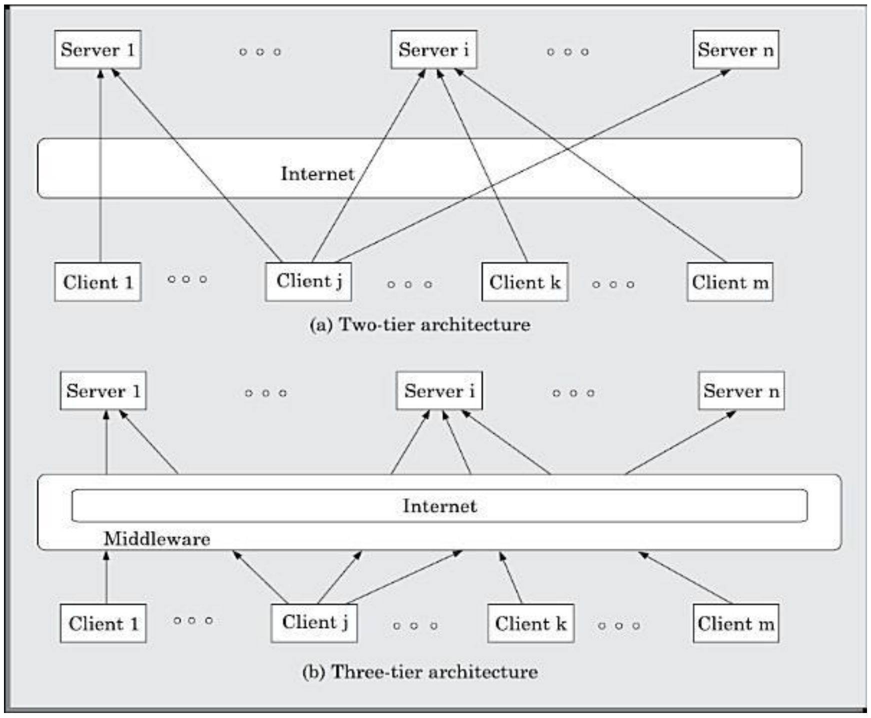


Figure 15.1: Two-tier and three-tier client-server architectures.

Functions of middleware
The important activities of the middleware include the following: The middleware keeps track of the addresses of servers. Based on a client request, it can therefore easily locate the required server.
It can translate between client and server formats of data and vice versa.
Two popular middleware standards are:

Common Object Request Broker Architecture (CORBA)
COM/DCOM

CORBA is being promoted by Object Management Group (OMG), a consortium of a large number of computer industries such as IBM, HP, Digital, etc. However, OMG is not a standards body. OMG in fact does not have any authority to make or enforce standards. It just tries to popularize good solutions with the hope that if a solution becomes highly popular, it would ultimately become a standard. COM/DCOM is being promoted mainly by Microsoft. In the following subsections, we discuss these two important middleware standards.

CORBA

Common object request broker architecture (CORBA) is a specification of a standard architecture for middleware. Using a CORBA implementation,a client can transparently invoke a service of a server object, which can be on the same machine or across a network. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and errorhandling; parameter marshalling and demarshalling; and operation dispatching.

CORBA Reference Model
The CORBA reference model has been shown in Figure 15.2. In the following subsection, we briefly discuss the major components of the CORBA reference model.

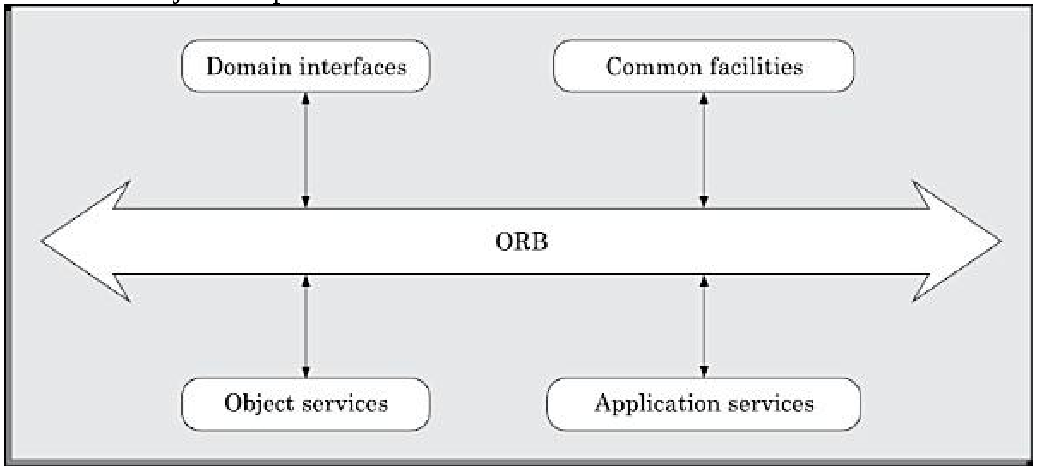


Figure 15.2: CORBA reference model.

ORB
ORB is also known as the **object bus**, since ORB supports communication among the different components attached to it. This is akin to a bus on a printed circuit board (PCB) on which the different hardware components (ICs) communicate. Observe that due to this analogy, even the symbol of a bus from the hardware domain is used to represent ORB (see Figure 15.2). The ORB handles client requests for any service, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the required server object is located, its programming language, its operating system or any other aspects that are not part of an object’s interface.
Domain interfaces
T h e s e interfaces provide services pertaining to specific application domains. Several domain services have been in use, including manufacturing, telecommunication, medical, and financial domains.

Object services
These are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of o bject services that fulfill this role are the following:
Naming Service: This allows clients to find objects based on names. Naming service is also called white page service.

Trading Service: This allows clients to find objects based on their properties. Trading service is also called yellow page service. Using trading service a specific service can be searched. This is akin to searching a service such as automobile repair shop in a yellow page directory.
There can be other services which can be provided by object services such as security services, life-cycle services and so on.

Common facilities

Like object service interfaces, these interfaces are also horizontally oriented, but unlike object services they are oriented towards end-user applications. An example of such a facility is the distributed document component facility (DDCF), a compound document common facility based on OpenDoc. DDCF allows for the presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.

Application interfaces

These are interfaces developed specifically for a given application.

CORBA ORB Architecture

The representation of Figure 15.3 is simplified since it does not show the various components of ORB. Let us now discuss the important components of CORBA architecture and how they operate. The ORB must support a large number of functions in order to operate consistently and effectively. In the carefully thought-out design of ORB, the ORB implements much of these functionality as pluggable modules to simplify the design and implementation of ORB and to make it efficient.

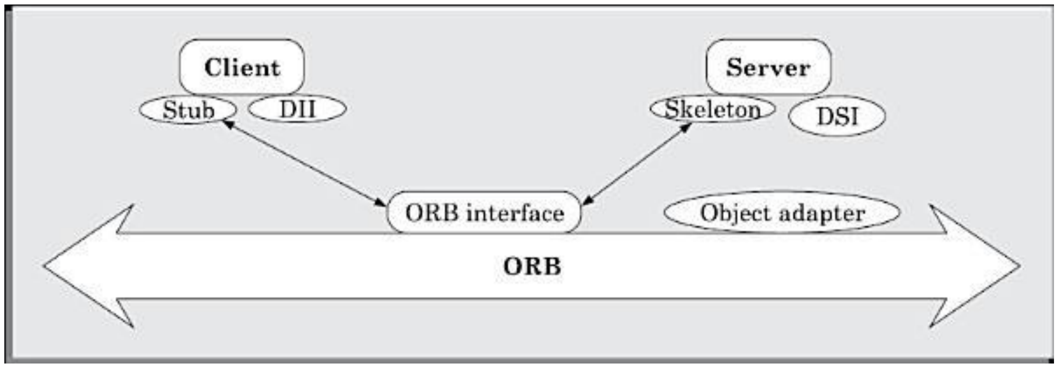


Figure 15.3: CORBA ORB architecture.

ORB

CORBA's most fundamental component is the object request broker (ORB) whose task is to facilitate communication between objects. The main responsibility of ORB is to transmit the client request to the server and get the response back to the client. ORB abstracts out the complexities of service invocation across a network and makes service invocation by client seamless and easy. The ORB simplifies distributed programming by decoupling clients from the details of the service calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller. ORB allows objects to hide their implementation details from clients. The different aspects of a program that are hidden (abstracted out) from the client include programming language, operating system, host hardware,and object location.

Stubs and skeletons

Using a CORBA implementation clients can communicate to the server in two ways—by using stubs or by using dynamic invocation interface (DII). The stubs help static service invocation,

where a client requests for a specific service using the required parameters. In the dynamic service invocation, the client need not know before hand about the required parameters and these are determined at the run time. Though dynamic service invocation is more flexible, static service invocation is more efficient that dynamic service invocation.
Service invocation by client through stub is suitable when the interface between the client and server is fixed and it does not change with time. If the interface is known before starting to develop client and the server parts then stubs can effectively be used for service invocation. The stub part resides in the client computer and acts as a proxy for the server which may reside in the remote computer. That is the reason why stub is also known as a proxy.

Object adapter
Service invocation through dynamic invocation interface (DII) transparently accesses the interface repository (OA). When an object gets created, it registers information about itself with OA. DII gets the relevant information from the IR and lets the client know about the interface being used.

CORBA Implementations
There are several CORBA implementations that are available for use. The following are a few popular ones.
Visibroker is a software from Borland is probably the most popular CORBA implementation. Netscape browser supports Visibroker.
Therefore, CORBA applications can be run using Netscape web browser. In other words, Netscape browser can act as a client for CORBA applications. Netscape is extremely popular and there are several millions of copies installed on desktops across the world. Orbix from Iona technologies. Java IDL.

Software Development in CORBA
Let us examine how software can be developed in CORBA. Before developing a client-server application, the solution is split into two parts —the client part and the serv part. Next, the exact client and server interfaces are determined. To specify an interface, interface definition language (IDL) is used. IDL is very similar to C++ and Java except that it has no executable statements. Using IDL only data interface between clients and servers can be defined. It supports inheritance so that interfaces can be reused in the same or across different applications. It also supports exception.
After the client-server interface is specified in IDL, an IDL compiler is used to compile the IDL specification. Depending on whether the target language in which the application is to be developed is Java, C++, C, etc., Different IDL compilers such as IDL2Java, IDL2C++, IDL2C etc. can be used as required. When the IDL specification is compiled, it generates the skeletal code for stub and skeleton. The stub and skeleton contain interface definitions and only the method body needs to be written by the programmers developing the components.
Inter-ORB communication
Initially, CORBA could only integrate components running on the same LAN. However, on certain applications, it becomes necessary to run the different components of the application in different networks. This shortcoming of CORBA 1.X was removed by CORBA 2.0. CORBA 2.0 defines general interoperability standard. The general inter-orb protocol (GIOP) is an abstract meta-protocol. It specifies a standard transfer syntax and a set of message formats for object

requests. The GIOP is designed to work over many different transport protocols. In a distributed implementation, every ORB must support GIOP mapped onto its local transport. GIOP can be used by almost any connection-oriented byte stream transport.
GIOP is popularly implemented on TCP/IP known as internet inter-ORB protocol (IIOP).

COM/DCOM

COM

The main idea in the component object model (COM) is that different vendors can sell binary components. Application can be developed by integrating off-the-shelf components. COM can be used to develop component applications on a single computer. The concepts used are very similar to CORBA. The components are known as binary objects.
These can be generated using languages such as Visual Basic, Delphi, Visual C++ etc. These languages have the necessary features to create COM components. COM components are binary objects and they exist in the form of either .exe or .dll (dynamic link library). The .exe components have separate existence. But .dll COM components are in process servers, that get linked to a process. For example, ActiveX is a dll type server, which gets loaded on the client-side.

DCOM

Distributed component object model (DCOM) is the extension of the component object model (COM). The restriction that clients and servers reside in the same computer is relaxed here. So, DCOM can operate on networked computers. Using DCOM, development is easy as compared to CORBA. Much of the complexities are hidden from the programmer.

SERVICE-ORIENTED ARCHITECTURE (SOA)

Service-orientation principles have their roots in the object-oriented designing. Many claim that service-orientation will replace object orientation; others think that the two are complementary paradigms.
SOA views software as providing a set of services. Each service composed of smaller services. Let us first understand what are software services. Services are implemented and provided by a component for use by an application developer. A service is a contractually de fined behaviour. That is, a component providing a service guarantees that its behaviour is as per the specifications. A few examples of services are the following—Filling out an online application, viewing an on-line bank-statement, and placing an online booking. Different services in an application communicate with each other.

The services are self-contained. That is, a service does not depend on the context or state of the other service. An application integrating different services works within a distributed-system architecture.
The main idea behind SOA is to build applications by composing software services.
SOA principally leverages the Internet and emerging the standardizations on it for interoperability among various services. An application is built using the services available on the Internet, and writing only the missing ones.
There are several similarities between services and components, which are as follows:
Reuse: Both a component and a service are reused across multiple applications.
Generic: The components and services are usually generic enough to be useful to a wide range of applications.

Composable: Both services and components are integrated together to develop an application.
Encapsulated: Both components and services are non-investigable through their interfaces.
Independent development and versioning: Both components and services are developed independently by different vendors and also continue to evolve independently.
Loose coupling: Both applications developed using the component paradigm and the SOA paradigm have loose coupling inherent to them. However, there are several dissimilarities between the components and the SOA paradigm, which are as follow:
The granularity (size) of services in the SOA paradigm are often 100 to 1,000 times larger than the components of the component paradigm.
Services may be developed and hosted on separate machines.
Normally components in the component paradigm are procured for use as per requirement (ownership). On the other hand, services are usually availed in a pay per use arrangement.
Instead of services embedding calls to each other in their source code, services use well-defined protocols which describe how services can talk to each other. This architecture facilitates a business process expert to tailor an application as per requirement. To meet a new business requirement, the business process expert can link and sequences services in a process known as orchestration.
SOA targets fairly large chunks of functionality to be strung together to form new services. That is, large services can be developed by integrating existing software services. The larger the chunks, the fewer the interfacings required. This leads to faster development. However, very large chunks may prove to be difficult to reuse.

Service-oriented Architecture (SOA): Nitty Gritty

The SOA paradigm utilises services that may be hosted on different computers. The different computers and services may be under the control of different owners. To facilitate application development, SOA must provide a means to offer, discover, interact with and use capabilities of the services to achieve desired results.

SOA involves statically and dynamically plugging-in services to build software. SOA players—BEA Aqua logic, Oracle Web services manager, HP Systinet Registry, MS .Net, IBM Web Sphere, Iona Artrix, Java composite application suite. Web services can be used to implement a service-oriented architecture. Web services can make functional building blocks accessible over standard Internet protocols independent of platforms and programming languages.

One of the central assumptions of SOA is that once a market place for services develops, services can be purchased to develop new applications. To build an application, one would use off-the-shelf services and possibly build some. When services are used across a large number of applications, automatically quality would improve and also price would reduce. When a service is used by a very large number of applications, the cost of using that service becomes near zero. Thus the cost of creating an application that uses widely used services would also be near zero, as all of the software services required would already exist and cost near zero, only orchestration of these services would be required to produce the application.

SOFTWARE AS A SERVICE (SAAS)

Owning software is very expensive. For example, a Rs. 50 Lakh software running on an Rs. 1 Lakh computer is common place. As with hardware, owning software is the current tradition across individuals and business houses. Most of IT budget now goes in supporting the software assets.

The support cost includes annual maintenance charge (AMC), keeping the software secure and virus free, and taking regular back-ups, etc.

But, often the usage of a specific software package does not exceed a couple of hours of usage per week. In this situation, it would be economically worthwhile to pay per hour of usage. This would also free the user from the botherance of maintenance, upgradation, backup, etc.

This is exactly what is advocated by SaaS. In this context, SaaS makes a case for pay per usage of software rather than owning software for use. SaaS is a software delivery model and involves customers to pay for any software per unit time of usage, with the price reflecting market place supply and demand.

As we can see, SaaS shifts "ownership" of the software from the customer to a service provider. Software owner provides maintenance, daily technical operation, and support for the software. Services are provided to the clients on amount of usage basis. The service provider is a vendor who hosts the software and lets the users execute on-demand charges per usage units. It also shifts the responsibility for hardware and software management from the customer to the provider. The cost of providing software services reduces as more and more customers subscribe to the service. Elements of outsourcing and application service provisioning are implicit in the SaaS model. Also, it makes the software accessible to a large number of customers who cannot afford to purchase the software outright. Target the "long tail" of small customers.

If we compare SaaS to SOA, we can observe that SaaS is a software delivery model, whereas SOA is a software construction model. Despite significant differences, both SOA and SaaS espouse closely related architecture models. SaaS and SOA complement each other. SaaS helps to offer components for SOA to use. SOA helps to help quickly realise SaaS. Also, the main enabler of SaaS and SOA are the Internet and web services technologies.