LECTURENOTES

SOFTWARE ENGINEERING

B.Tech, 6THSemester,CSE

Prepared by:

MRS SANJUKTA URMA

Lecturer in Computer Science & Engineering

Vikash Institute of Technology Bargarh

(Approved by AICTE, New Delhi & Affiliated to BPUT, Odisha)

BarahagudaCanalChowk,Bargarh,Odisha-768040

www.vitbargarh.ac.in

DISCLAIMER

- This document does not claim any originality and cannot be used as a substitute for prescribed textbooks.
- The information presented here is merely a collection by Mrs. SANJUKTA URMA with the inputs of students for their respective teaching assignments as an additional tool for the teaching- learning process.
- Various sources as mentioned at the reference of the document as well as freely available materials from internet were consulted for preparing this document.
- Further, this document is not intended to be used for commercial purpose and the authors are not accountable for any issues, legal or otherwise, arising out of use of this document.
- The author makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaims any implied warranties of merchantability or fitness for a particular purpose.

****

COURSECONTENT

SOFTWARE ENGINEERING

B.Tech, 6THSemester, CSE

➢ Module-I: Software Process Models:

Software Product, Software crisis, Handling complexity through Abstraction and Decomposition, Overview of software development activities, Process Models, Classical waterfall model, iterative waterfall model, prototyping mode, evolutionary model, spiral model, RAD model, Agile models: Extreme Programming, and Scrum.

➢ Module-II: Software Requirements Engineering

Requirement Gathering and Analysis, Functional and Non-functional requirements, Software Requirement Specification (SRS), IEEE 830 guidelines, Decision tables and trees. Structured Analysis & Design: Overview of design process, High- level and detailed design, Cohesion and coupling, Modularity and layering, Function–Oriented software design: Structured Analysis using DFD Structured Design using Structure Chart, Basic concepts of Object Oriented Analysis & Design. User interface design, Command language, menu and iconic interfaces

➢ Module-III Coding and Software Testing Techniques

Coding, Code Review, documentation. Testing: - Unit testing, Black-box Testing, White-box testing, Cyclomatic complexity measure, coverage analysis, mutation testing, Debugging techniques, Integration testing, System testing, Regression testing. Software Reliability and Software

➢ Module- Maintenance:

Basic concepts in software reliability, reliability measures, reliability growth modelling, Quality SEI CMM, Characteristics of software maintenance, software reverse engineering, software reengineering, software reuse. Emerging Topics: Client-Server Software Engineering, Service-oriented Architecture (SOA), and Software as a Service (SaaS)

REFERENCES

SOFTWARE ENGINEERING

B.Tech, 6THSemester, CSE

Books:

1. Fundamentals of Software Engineering, Rajib Mall , 5th Ed, PHI, 2018.
2. Software Engineering, A Practitioner's Approach, Roger S. Pressman, 8th Ed, TMG H 2019
3. Software Engineering, I. Sommerville, 9th Ed., Pearson Education, 2011

Digital Learning Resources:

Course Name:	Software Engineering
Course Link:	https://nptel.ac.in/courses/106/105/106105182/
Course Instructor:	Prof. Rajib Mall, IIT Kharagpur
Course Name:	Software Engineering
Course Link:	https://nptel.ac.in/courses/106/101/106101061/
Course Instructor:	Prof. N.L. Sarda, Prof. R. K Joshi, Prof. U. Bellur IIT Bombay

MODULE-I

INTRODUCTION

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

What is software engineering?

A popular definition of software engineering is: "A systematic collection of good program development practices and techniques". Good program development techniques have resulted from research innovations as well as from the lessons learnt by programmers through years of programming experiences. An alternative definition of software engineering is: "An *engineering approach* to develop software". Based on these two point of views, we can define software engineering as follows: Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Software Product-

Software Products are nothing but software systems delivered to the customer with the documentation that describes how to install and use the system. In certain cases, software products may be part of system products where hardware, as well as software, is delivered to a customer. Software products are produced with the help of the software process. The software process is a way in which we produce software.

Types of Software Products

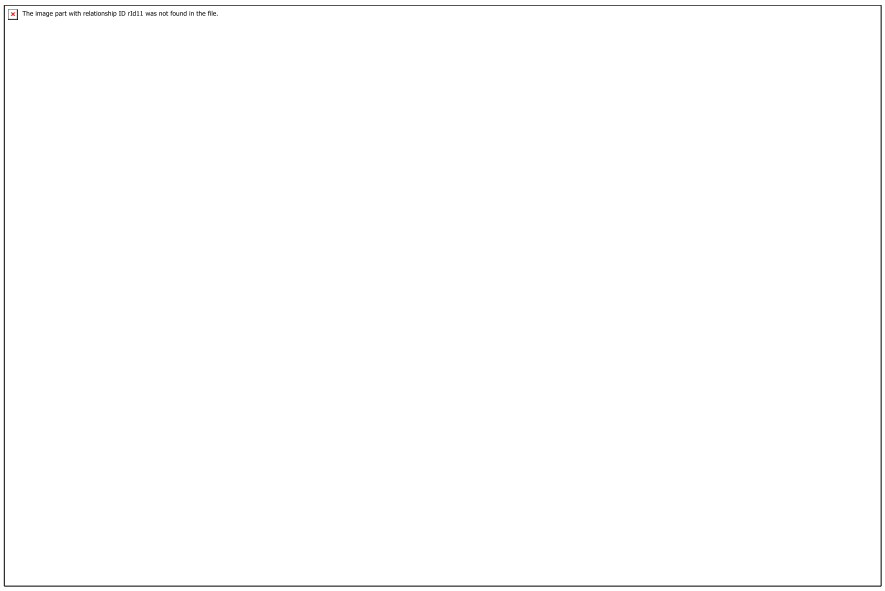
Software products fall into two broad categories:

Generic products: Generic products are stand-alone systems that are developed by a production unit and sold on the open market to any customer who can buy them.

Customized Products: Customized products are the systems that are commissioned by a particular customer. Some contractor develops the software for that customer.

Characteristics of Software Product

A well-engineered software product should possess the following essential characteristics:



Characteristics of Software Product

Efficiency: The software should not make wasteful use of system resources such as memory and processor cycles.

Maintainability: It should be possible to evolve the software to meet the changing requirements of customers.

Dependability: It is the flexibility of the software that ought to not cause any physical or economic injury in the event of system failure. It includes a range of characteristics such as reliability, security, and safety.

In time: Software should be developed well in time.

Within Budget: The software development costs should not be overrun, and they should be within the budgetary limit.

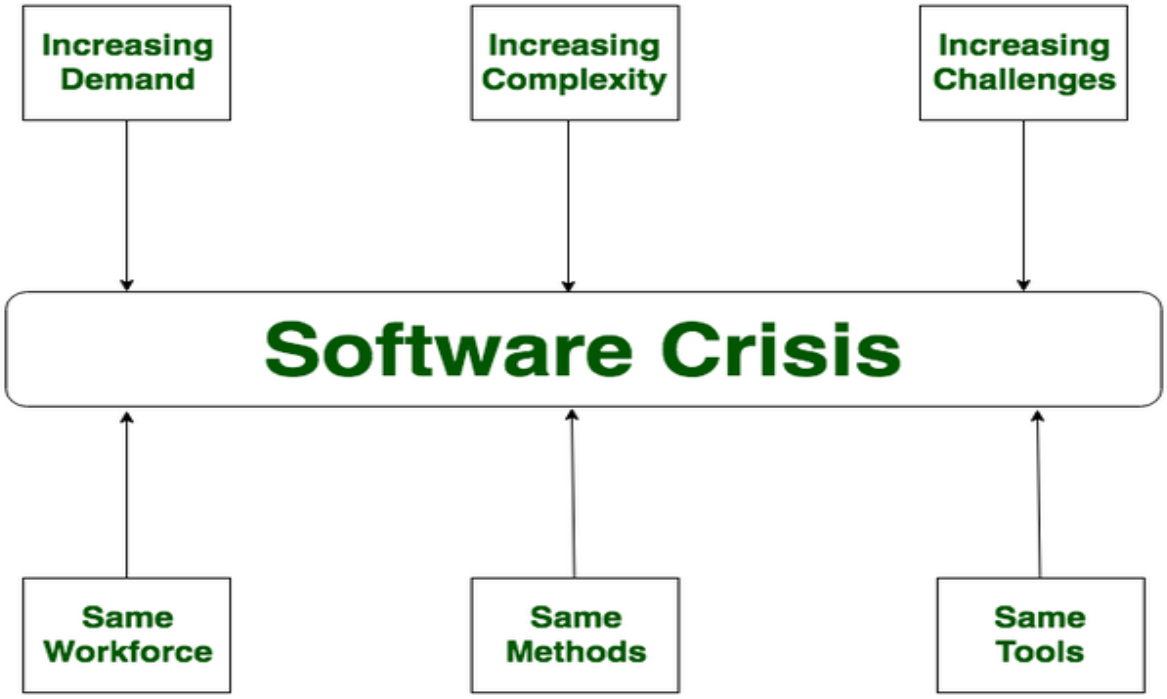
Functionality: The software system should exhibit the proper functionality, i.e., it should perform all the functions it is supposed to perform.

Adaptability: The software system should have the ability to adapted to a reasonable extent with the changing requirements.

Software Crisis-

Software Crisis is a term used in computer science for the difficulty of writing useful and efficient computer programs in the required time. The software crisis was due to using the same workforce, same methods, and same tools even though rapidly increasing software demand, the complexity of software, and software challenges. With the increase in software complexity, many software problems arose because existing methods were insufficient.

Suppose we use the same workforce, same methods, and same tools after the fast increase in software demand, software complexity, and software challenges. In that case, there arise some issues like software budget problems, software efficiency problems, software quality problems, software management, and delivery problems, etc. This condition is called a Software Crisis.



Causes of Software Crisis

Following are the causes of Software Crisis:

- The cost of owning and maintaining software was as expensive as developing the software.
- At that time Projects were running overtime.
- At that time Software was very inefficient.

- The quality of the software was low quality.
- Software often did not meet user requirements.
- The average software project overshoots its schedule by half.
- At that time Software was never delivered.

Non-optimal resource utilization.

- Challenging to alter, debug, and enhance.
- The software complexity is harder to change.

A Solution to the Software Crisis

There is no single solution to the crisis. One possible solution to a software crisis is Software Engineering because software engineering is a systematic, disciplined, and quantifiable approach. For preventing software crises, there are some guidelines:

- Reduction in software over budget.
- The quality of the software must be high.
- Less time is needed for a software project.
- Experienced and skilled people working on the software project.
- Software must be delivered.
- Software must meet user requirements.

SOFTWARE DEVELOPMENT PROJECTS

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

Programs versus Products

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

maintenance manual, design document, test documents, etc., we call this toy software as *programs*.

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

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

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.

Abstraction

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

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

When using the principle of abstraction to understand a complex problem, we focus our attention on only one or two specific aspects of the problem and ignore the rest. Whenever we omit some details of a problem to construct an abstraction, we construct a *model* of the problem. In everyd ay 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 soon. On the other hand, the *political map* shows states, capitals, and national boundaries, etc. The physical map is an abstract model of the country and ignores the state and district boundaries. The political map, on the other hand, is another abstraction of the country that ignores the physical characteristics such as elevation of lands, vegetation, etc. It can be seen that, for the same object (e.g. country), several abstractions are possible. In each abstraction, some aspects of the object is ignored. We understand a problem by abstracting out different aspects of a problem (constructing different types of models) and understanding them. It is not very difficult to realize that proper use of the principle of abstraction can be a very effective help to master even intimidating problems.

- Consider the following situation. Suppose you are asked to develop an understanding of all the living beings inhabiting the earth. If you use the naive approach, you would start taking up one living being after another who inhabit the earth and start understanding them. Even after putting in tremendous effort, you would make little progress and left confused since there are billions of living things on earth and the information would be just too much for any one to handle. Instead, what can be done is to build and understand an abstraction hierarchy of all living beings as shown in Figure 1.7. 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.

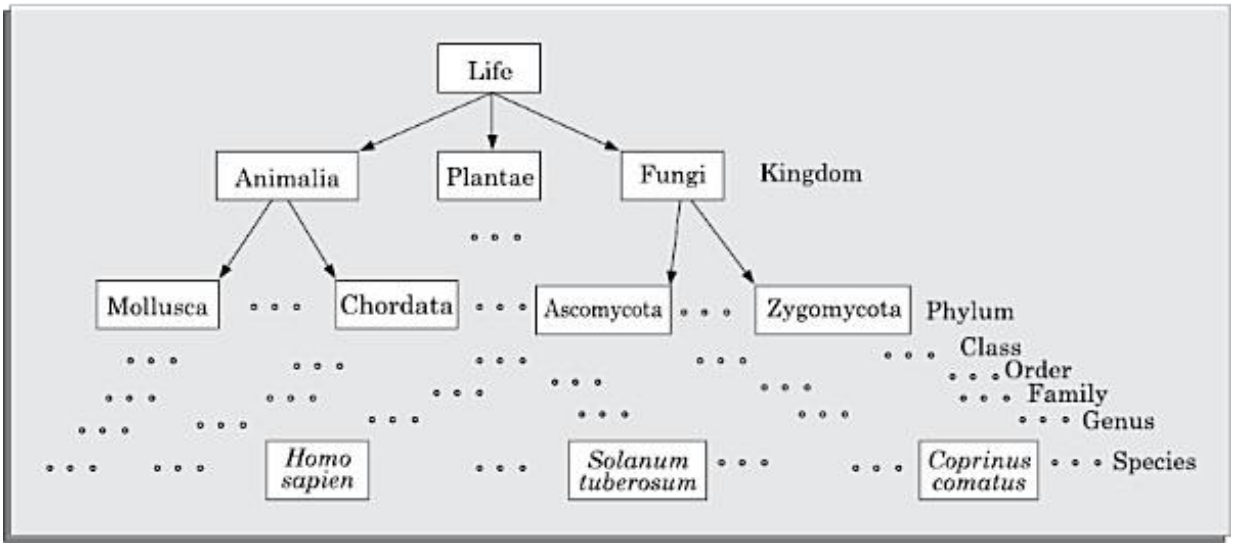


Figure 1.7: An abstraction hierarchy classifying living organisms.

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. 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 o f a use of the principle of decomposition, consider the following. You would understand a book better when the contents are decomposed (organized) 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.

SOFTWARE LIFE CYCLE MODELS

We discussed a few basic issues in software engineering. We pointed out a few important differences between the exploratory program development style and the software engineering approach. Please recollect from our discussions in Chapter 1 that the exploratory style is also known as the *build* and *fix* programming. In build and fix programming, a programmer typically starts to write the program immediately after he has formed an informal understanding of the requirements. Once program writing is complete, he gets down to fix anything that does not meet the user's expectations. Usually, a large number of code fixes are required even for toy programs. This pushes up the development costs and pulls down the quality of the program. Further, this approach usually turns out to be a recipe for project failure when used to develop non-trivial programs requiring team effort. In contrast to the build and fix style, the software engineering approaches emphasize software development through a well-defined and ordered set of activities. These activities are graphically modelled (represented) as well as textually described and are variously called a s *software life cycle model*, *software development life cycle* (SDLC) *model*, and *software development process model*. Several life cycle models have so far been proposed. However, in this Chapter we confine our attention to only a few important and commonly used ones.

In this chapter, w e first discuss a few basic concepts associated with life cycle models. Subsequently, we discuss the important activities that have been prescribed to be carried out in the classical waterfall model. This is intended to provide an insight into the activities that are carried out as part of every life cycle model. In fact, the classical waterfall model can be considered as a basic model and all other life cycle models as extensions of this model to cater to specific project situations. After discussing the waterfall

model, we discuss a few derivatives of this model. Subsequently we discuss the spiral model that generalizes various life cycle models. Finally, we discuss a few recently proposed life cycle models that are categorized under the umbrella term agile model. Of late, agile models are finding increasing acceptance among developers and researchers.

oftware 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.

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, analyze 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 methodologies 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:

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. Besides, we shall see later in this text that this model though not used for software development; is implicitly used while documenting software.
The classical waterfall model divides the life cycle into a set of phases as

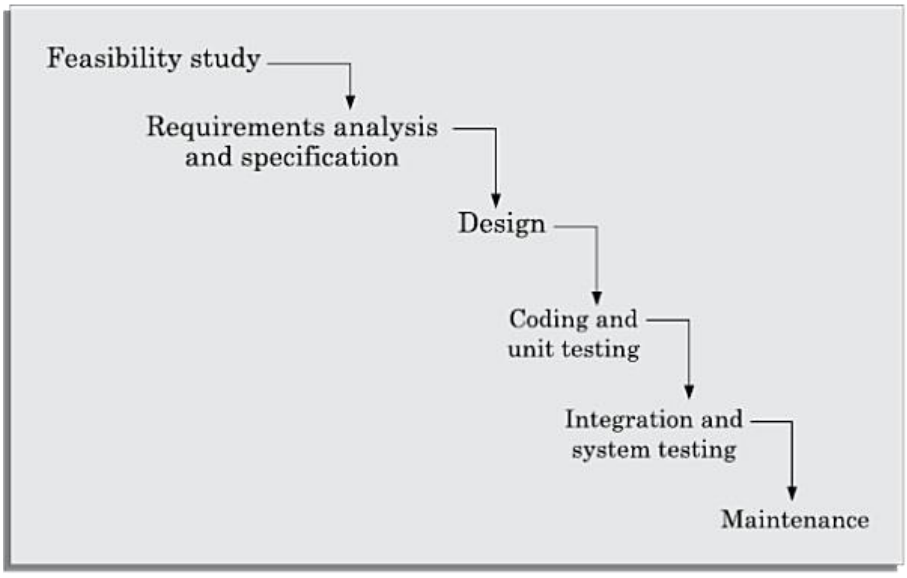


Figure 2.1: Classical waterfall model.

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.

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. It needs to be kept in mind that some of the text

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.

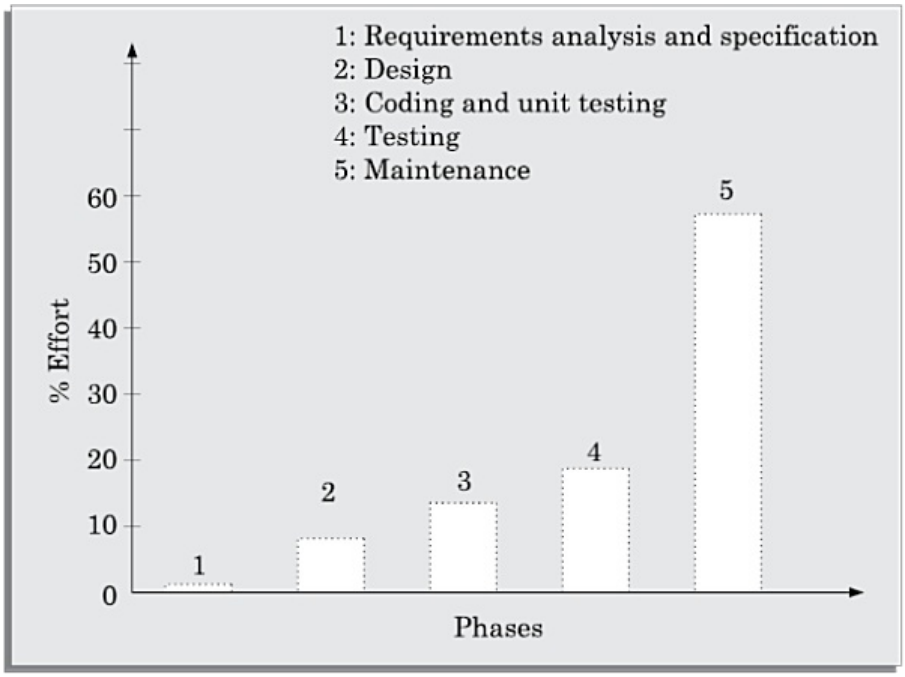


Figure 2.2: Relative effort distribution among different phases of a typical product.

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. Thefeasibility 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.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document

them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification. In the following subsections, we give an overview of these two activities:

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. For this, first requirements are gathered from the customer and then the gathered requirements are analysed. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements. Note that a n *inconsistent* requirement is one in which some part of the requirement contradicts with some other part. On the other hand, a n *incomplete* requirement is one in which some parts of the actual requirements have been omitted.
- **Requirements specification:** After the requirement gathering and analysis activities are complete, the identified requirements are documented. This is called a *software requirements specification* (SRS) document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer. Therefore, understandability of the SRS document is an important issue. The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer. The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it. In Chapter 4, we examine the requirements analysis activity and various issues involved in developing a good SRS document in more detail.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the softwre 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. In the following, we briefly discuss the essence of these two 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 of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Various modules making up a software are almost never integrated in one shot (can you guess the reason for this?). Integration of various modules are normally carried out incrementally over a number of steps.

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

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

- -testing: testing is the system testing performed by the development team.
- -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.

Shortcomings of the classical waterfall model

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

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

The classical waterfall model is idealistic in the sense that it assumes that no error is ever committed by the developers during any of the life cycle phases, and therefore,

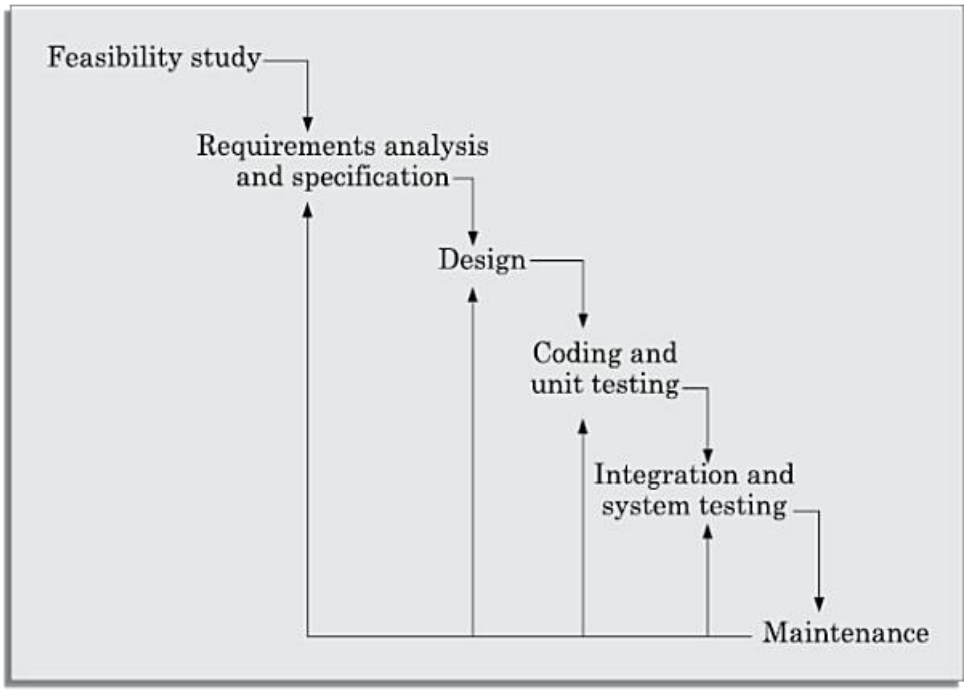
incorporates no mechanism for error correction.

terative Waterfall Model

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

The 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.

Figure 2.3: Iterative waterfa l model.



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

Phase containment of errors

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

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.

● time would be as shown in Figure 2.4.

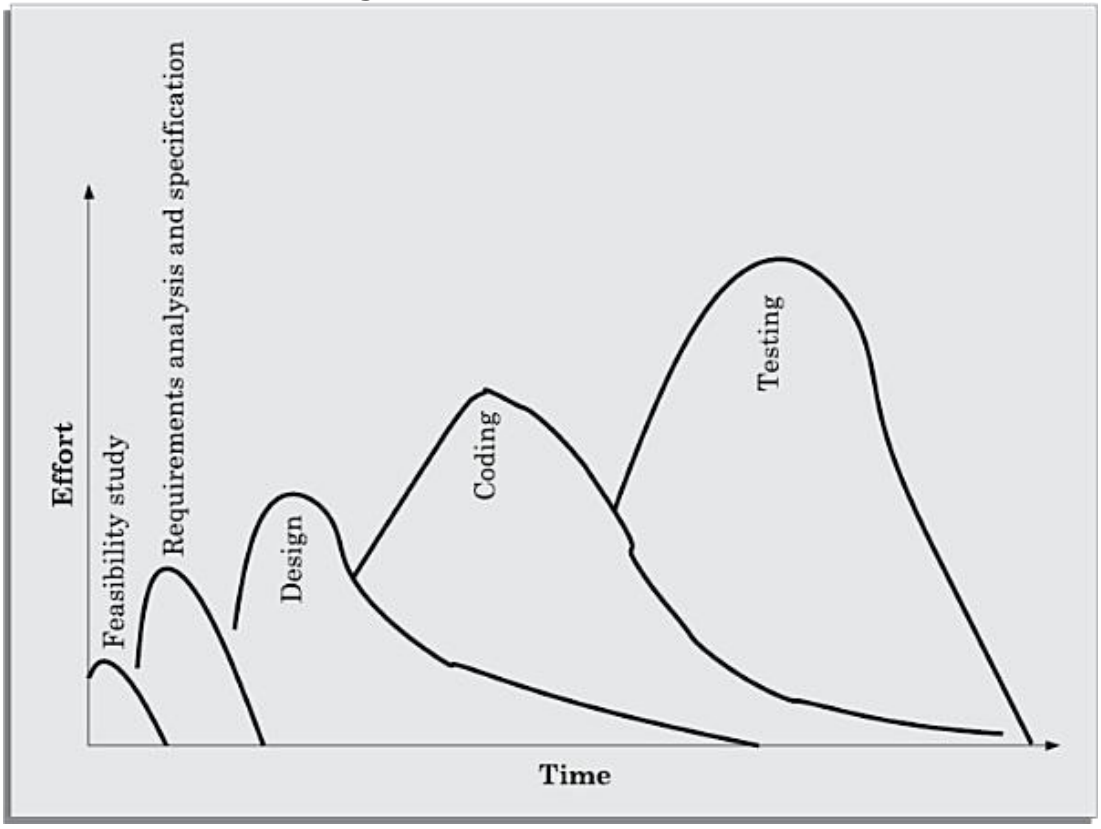


Figure 2.4: Distribution of effort for various phases in the iterative waterfa l model.

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

Prototyping Model

The prototype model is also a popular life cycle model. The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working prototype of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, o r inefficient performance as compared to the actual software. A prototype can be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. Normally the term rapid prototyping is used when software tools are used for prototype construction. For example, tools based on fourth generation languages (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

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

- It is advantageous to use the prototyping model for development of the graphical user interface (GUI) part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer. This is a valuable mechanism for gaining better understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the graphical user interface (GUI) part of a system. For the user, it becomes much easier to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a hypothetical user interface.

The GUI part of a software system is almost always developed using the prototyping model.

- The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development. For example, consider a situation where the development team has to write a command language interpreter as part of a graphical user interface development. Suppose none of the team members has ever written a compiler

before. Then, this lack of familiarity with a required development technology is a technical risk. This risk can be resolved by developing a prototype compiler for a very small language to understand the issues associated with writing a compiler for a command language. Once they feel confident in writing compiler for the small language, they can use this knowledge to develop the compiler for the command language. Often, major design decisions depend on issues such as the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype is often the best way to resolve the technical issues.

- An important reason for developing a prototype is that it is impossible to "get it right" the first time. As advocated by Brooks [1975], one must plan to throw away the software in order to develop a good software later. Thus, the prototyping model can be deployed when development of highly optimised and efficient software is required.

From the above discussions, we can conclude the following:

The prototyping model is considered to be useful for the development of not only the GUI parts of a software, but also for a software project for which certain technical issues are not clear to the development team.

Life cycle activities of prototyping model

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

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

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

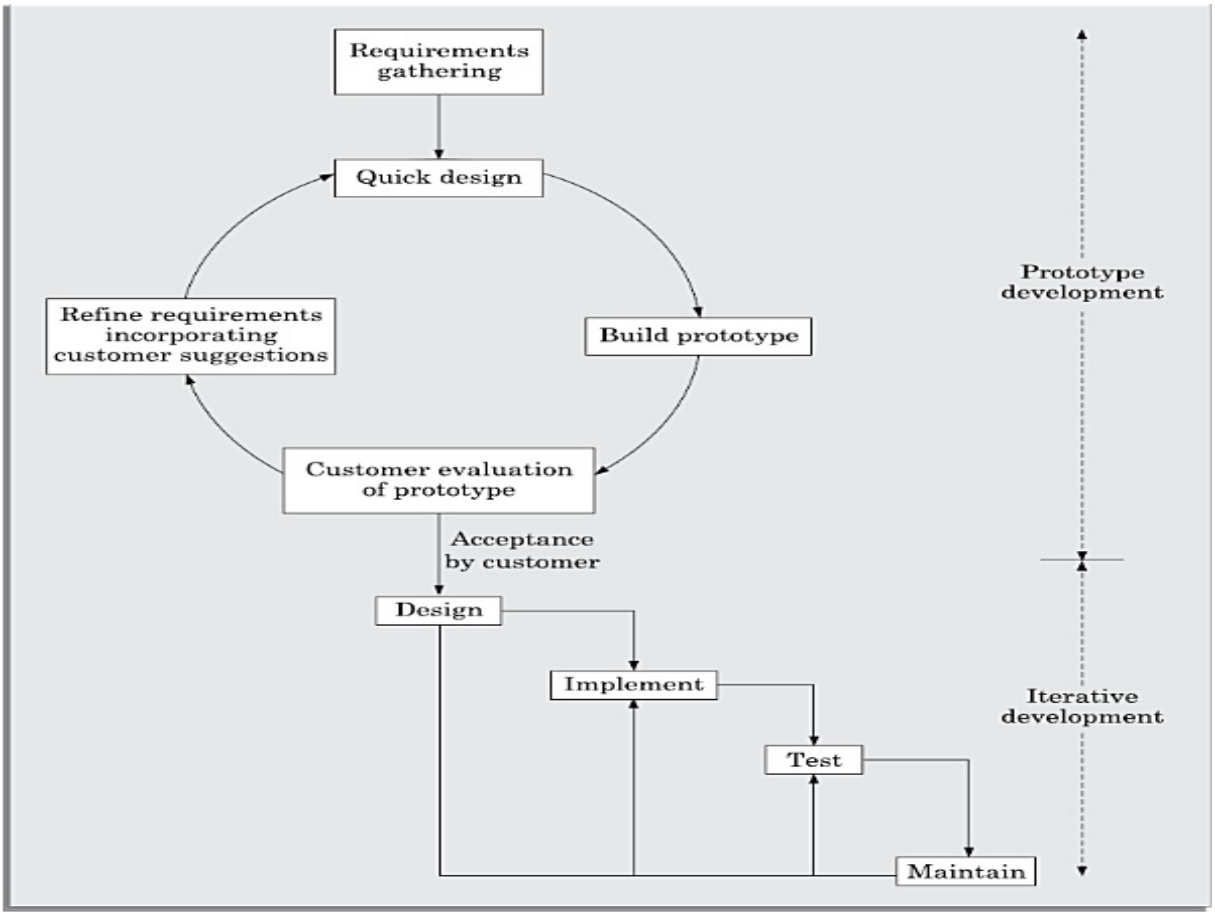


Figure 2.6: Prototyping model of software development.

trengths of the prototyping model

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

Weaknesses of the prototyping model

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

Incremental Development Model

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

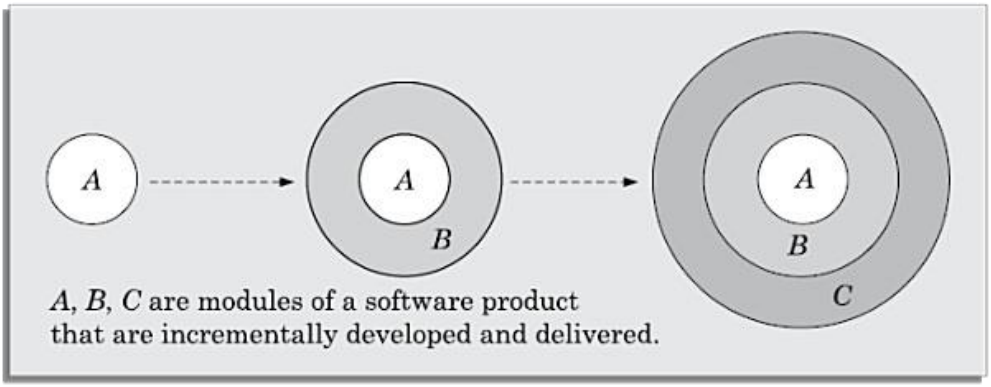


Figure 2.7: Incremental software development.

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted i n Figure 2.7. At any time, plan is made only for the next increment and no long-term plans a re made. Therefore, it becomes easier to accommodate change requests from the customers.The development team first undertakes to develop the core features of the system. The core or basic features are those that do not need to invoke any services from the other features. On the other hand, non-core features need services from the core features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in Figure 2.8. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.

requirements gathering and specification, the requirements are split into several versions. Starting with the core (version 1), in each successive increment, the next version is constructed using an iterative waterfall model of development and deployed at the customer site. After the last (shown as version n) has been developed and deployed at the client site, the full software is deployed.

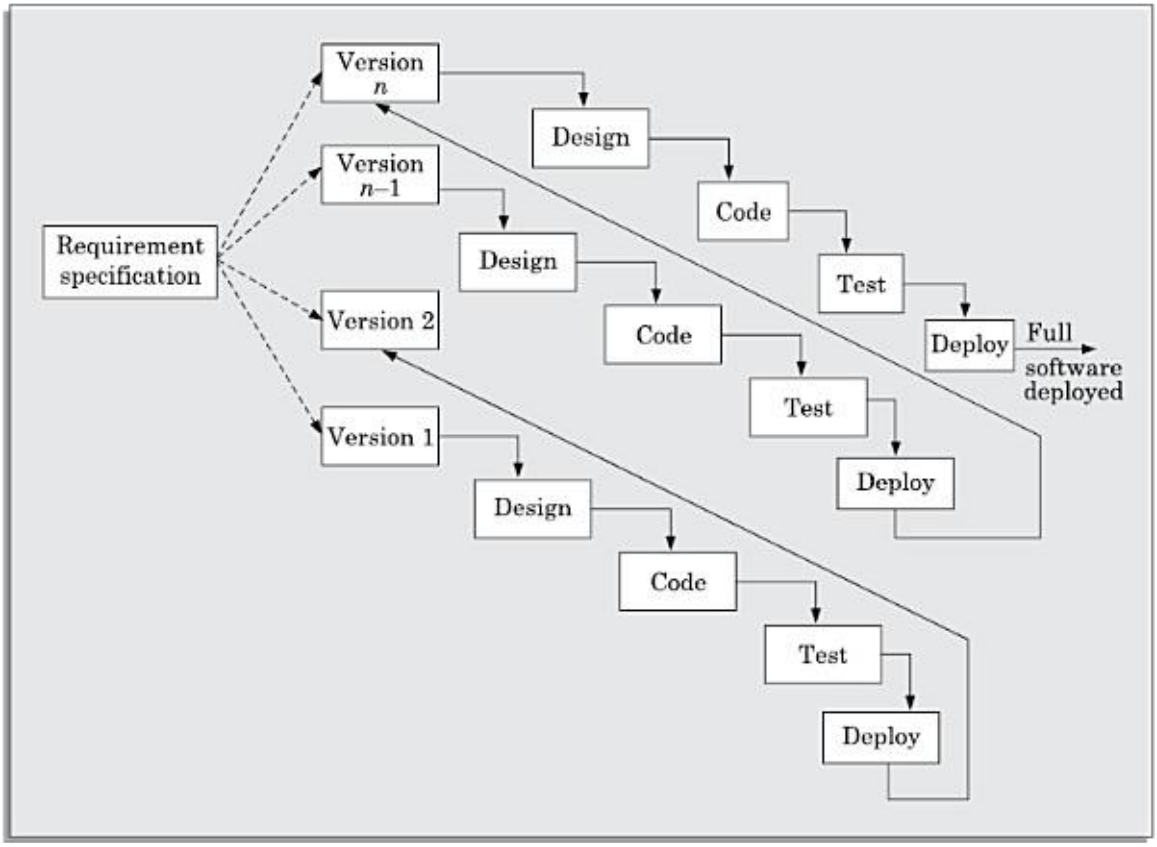


Figure 2.8: Incremental model of software development.

Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

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

Advantages

The evolutionary model of development has several advantages. Two importan advantages of using this model are the following:

Effective elicitation of actual customer requirements: In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.

Easy handling change requests: In this model, handling change requests is easier as no long term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

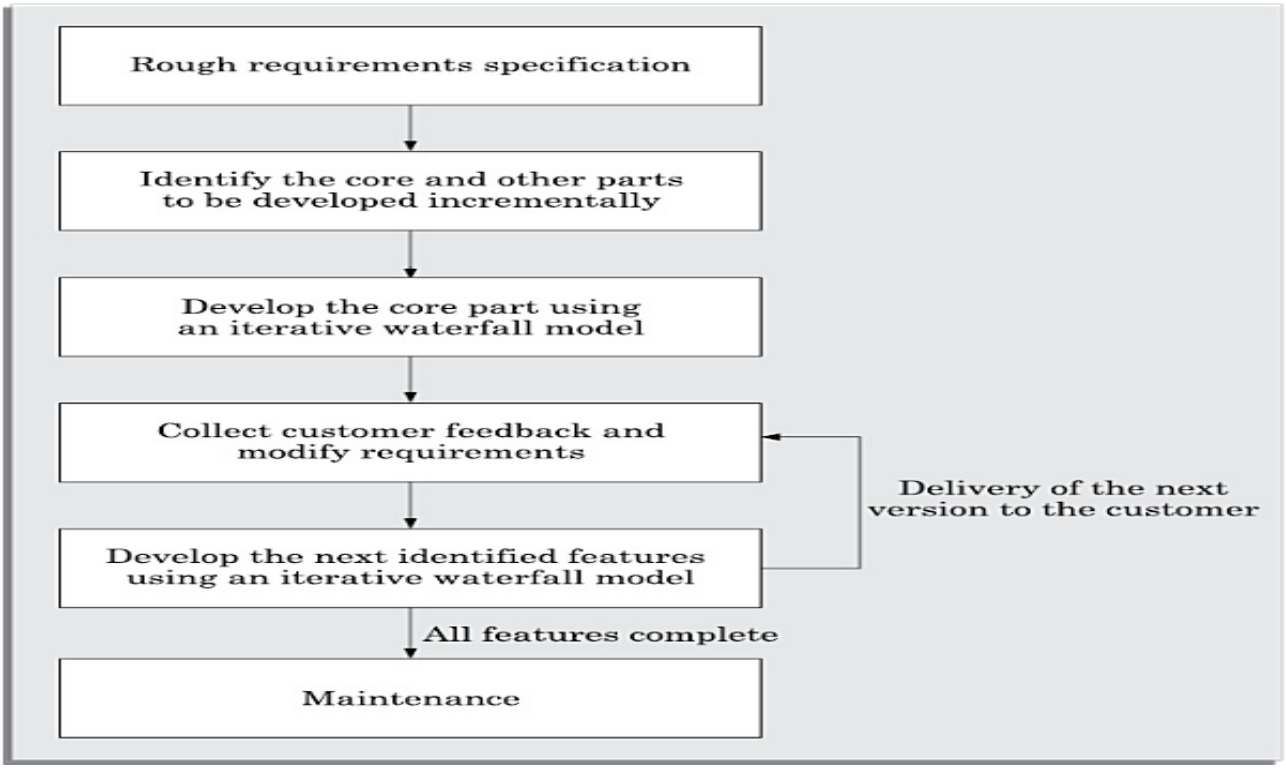
The main disadvantages of the successive versions model are as follows:

- Feature division into incremental parts can be non-trivial: For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered. Further, even for larger problems, often the features are so interwined and dependent on each other that even an expert would need

considerable effort to plan the incremental deliveries.

- Ad hoc design: Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

successively refined and feature-enriched until the full software is realised. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the incremental development model, complete requirements are first developed and the SRS document prepared. In contrast, in the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development



- Figure 2.9: Evolutionary model of software development

Applicability of the evolutionary model

The evolutionary model is normally useful for very large products, where it is easier t o find modules for incremental implementation. Often evolutionary model is used when the customer prefers to receive the product in increments so that he can start using the different features as and when they are delivered rather than waiting all the time for

the full product to be developed and delivered. Another important category of projects for which the evolutionary model is suitable, is projects using object-oriented development.

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

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

RAPID APPLICATION DEVELOPMENT (RAD)

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

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

The major goals of the RAD model are as follows:

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

Main motivation

In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start. However, often clients do not know what they exactly wanted until they saw a working system. It has now become well accepted among the practitioners that only through the process commenting on an installed application that the exact requirements can be brought out. But in the iterative waterfall model, the customers do not get to see the software, until the development is complete in all respects and the software has been delivered and installed. Naturally, the delivered software often does not meet the customer expectations and many change request are generated by the customer. The changes are incorporated through subsequent maintenance efforts. This made the cost of accommodating the changes extremely high and it usually took a long time to have a good solution

in place that could reasonably meet the requirements of the customers. The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed and refined prototypes.

Working of RAD

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

The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

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

- How does RAD facilitate faster development?
- The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.
- RAD model emphasizes code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialized tools to facilitate fast creation of working prototypes. These specialized tools usually support the following features:

- Visual style of development. Use of reusable components.

Applicability of RAD Model

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

- Customised software: As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software. For example, a company might have developed a software for automating the data processing activities at one or more educational institutes. When any other institute requests for an automation package to be developed, typically only a few aspects needs to be tailored—since among different educational institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records etc. are similar to a large extent. Projects involving such tailoring can be carried out speedily and cost- effectively using the RAD model.
- Non-critical software: The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the Iterative waterfall model may provide a better solution.
- Highly constrained pro ject schedule: RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
- Large software: Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

Application characteristics that render RAD unsuitable

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

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

Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

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

AGILE DEVELOPMENT MODELS

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present day software projects. The main difficulties included handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customised applications. Capers Jones carried out research involving 800 real- life software development projects, and concluded that on the average 40 per cent of the requirements is arrived after the development has already begun. In this context, over the last two decade or so, several life cycle models have been proposed to overcome the important shortcomings o f the waterfall- based models that become conspicuous when used in modern software development projects.

Over the last two decades or so, projects using iterative waterfall-based life cycle models are becoming rare due to the rapid shift in the characteristics of the software development projects over time. Two changes that are becoming noticeable are rapid shift from development of software products to development of customised software and the increased emphasis and scope for reuse.

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

- In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If at

all any later requirement changes becomes unavoidable, then the cost of accommodating it becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons.

- As pointed out in Chapter 1, over the last two decades or so, customized applications (services) has become common place and the sales revenue generated worldwide from services already exceeds that of the software products. Clearly, iterative waterfall model is not suitable for development of such software. Since customization essentially involves reusing most of the parts of an existing application and consists of only carrying out minor modifications by writing minimal amounts of code. For such development projects, the need for more appropriate development models was deeply felt, and many researchers started to investigate in this direction.
- Waterfall model is called a heavy weight model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.
- Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

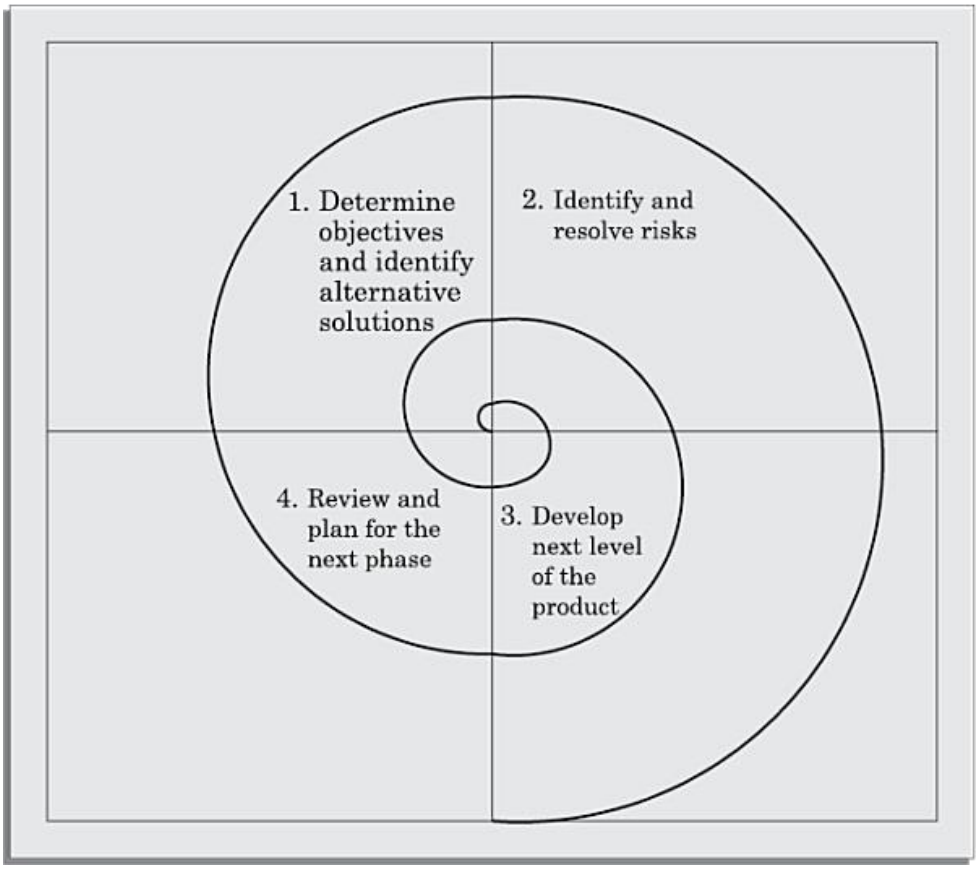
The agile software development model was proposed in the mid-1990s to overcome the serious shortcomings of the waterfall model of development identified above. The agile model was primarily designed to help a project to adapt to change requests quickly.1Thus, a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project, i.e. removing activities that may not be necessary for a specific project. Also, anything that that wastes time and effort is avoided.

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

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

In the agile model, the requirements are decomposed into many small parts that can be SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops (see Figure 2.10). The exact number of loops of the spiral is not fixed and can vary from project to project. The number of loops shown in Figure 2.10 is just an example. Each loop of the spiral is called a phase of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks. A prominent feature of the spiral model is handling unforeseen risks that can show up much after the project has started. In this context, please recollect that the prototyping model can be used effectively only when the risks in a project can be identified upfront before the



development work starts. As we shall discuss, this model achieves this by incorporating much more flexibility compared to SDLC other modelWhile the prototyping model does provide explicit support for risk handling, the risks are assumed to have been identified completely before the project start. This is required since the prototype is constructed only at the start of the project. In contrast, in the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation. Over each loop, one or more features of the product are elaborated and analysed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

Figure 2.10: Spiral model of software development

.

Risk handling in spiral model

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. If the data access rate is too slow, possibly a caching scheme can be implemented or a fastercommunication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

communication scheme can be deployed to overcome the slow data access rate. Such risk resolutions are easier done by using a prototype as the pros and cons of an alternate solution scheme can evaluated faster and inexpensively, as compared to experimenting using the actual software application being developed. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

2.1.1 Phases of the Spiral Model

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

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

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

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

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

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

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

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

Advantages/pros and disadvantages/cons of the spiral model

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

In spite of the disadvantages of the spiral model that we pointed out, for certain

categories of projects, the advantages of the spiral model can outweigh its disadvantages.

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

Spiral model as a meta model

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

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

2.2 A COMPARISON OF DIFFERENT LIFE CYCLE MODELS

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

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

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

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

The spiral model is considered a meta model and encompasses all other life cycle models. Flexibility and risk handling are inherently built into this model. The spiral

model is suitable for development of technically challenging and large software that are prone to several kinds of risks that are difficult to anticipate at the start of the project. However, this model is mu ch more complex than the other models—this is probably a factor deterring its use in ordinary projects.

MODULE-2

Requirements Analysis and Specification

- All plan-driven life cycle models prescribe that before starting to develop a software, the exact requirements of the customer must be understood and documented.
- Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, and thereby alarmingly pushes up the development costs.
- A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases.

Overview of 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 the software requirements specification (SRS) document.
- The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organize the requirements into a document called the Software Requirements Specification (SRS) document.

Who performs requirements analysis

- Requirements analysis and specification activity is usually carried out by a few experienced members of the development team.
- It normally requires them to spend some time at the customer site.
- The engineers who gather and analyze customer requirements and then write the requirements specification document are known as system analysts.

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

- Requirements gathering and analysis.
- Requirements specification.

Requirements gathering and analysis:

- The complete set of requirements are almost never available in the form of a single document from the customer.
- Complete requirements are rarely obtainable from any single customer representative.
- We can conceptually divide the requirements gathering and analysis activity into two separate tasks: Requirements gathering and 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.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- 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.
- Important ways in which an experienced analyst gathers requirements:

• 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 a statement of purpose (SoP) document to the developers.

• 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.

Refer to: Delphi method

• 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 software is also called a task.
- 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 realize the required functionality in consultation with the users.

- A task can have many scenarios of operation.
- The different scenarios of a task may take place when the task is invoked under different situations. For different types of scenarios of a task, the behavior of the software can be different.
- Form analysis:
- Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.
- In form analysis the existing forms and the formats of the notifications produced are analyzed to determine the data input to the system and the data that are output from the system.

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

Anomaly:

- An anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible.
- Example: While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder: "When the temperature becomes high, the heater should be switched off". Please note that words such as "high", "low", "good", "bad" etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted.

Inconsistency:

- Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
- Example: Consider the following two requirements that were collected from two different stakeholders in a process control application development project.
- The furnace should be switched-off when the temperature of the furnace rises above 500°C.
- When the temperature of the furnace rises above 500°C, the water shower should be switched-on and the furnace should remain on.

Incompleteness:

- An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much

● Example: In a chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200°C

● then an alarm bell must be sounded. However, on an examination of all requirements, it was found that there is no provision for resetting the alarm bell after the temperature has been brought down in any of the requirements. This is clearly an incomplete requirement.

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 organize 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.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.
- A well-formulated SRS document finds a variety of usage:
  - Forms an agreement between the customers and the developers.
  - Reduces future reworks.
  - Provides a basis for estimating costs and schedules
  - Provides a baseline for validation and verification
  - Facilitates future extensions

Users of SRS document:

- **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.
- **Software developers:**
  The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- **Test engineers:**
  The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working.
- **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.

Characteristics of a Good SRS Document:

- IEEE Recommended Practice for Software Requirements Specifications 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.
  - **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.
  - **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.
  - **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. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured.
  - **Identification of response to undesired events:**

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.

Categories of Customer requirements:

An SRS document should clearly document the following aspects of a software:

- Functional requirements
- Non-functional requirements
  - Design and implementation constraints
  - External interfaces required
  - Other non-functional requirements
- Goals of implementation.

Functional Requirements:

- The functional requirements capture the functionalities required by the users from the system.
- Consider a software as offering a set of functions $\{f_i\}$ to the user.
- These functions can be considered similar to a mathematical function f : I → O, meaning that a function transforms an element (i_i) in the input domain (I) to a value (o_i) in the output (O).
- In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements.
- Each high-level function is an instance of use of the system (use case) by the user in some way.

- A high-level function is one using which the user can get some useful piece of work done.
- Each high-level requirement typically involves accepting some data from the user through a user interface, transforming it to the required response, and then displaying the system response in proper format.
- A high-level function transforms certain input data to output data.
- Except for very simple high-level functions, a function rarely reads all its required data in one go and rarely outputs all the results in one shot.

- A high-level function usually involves a series of interactions between the system and one or more users.

- Functional requirements form the basis for most design and test methodologies.
- Unless the functional requirements are properly identified and documented, the design and testing activities cannot be carried out satisfactorily.
- Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented.
- A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data.
- ***Refer to Withdraw cash from ATM example in text book.***

Non-functional Requirements:

- The non-functional requirements are non-negotiable obligations that must be supported by the software.
- The non-functional requirements capture those requirements of the customer that cannot be expressed as functions.
- Aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput.
- The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.
- **Design and implementation constraints:**
    - Design and implementation constraints are an important category of non-functional requirements describing any items or issues that will limit the options available to the developers.
    - Some of the example constraints can be—corporate or regulatory policies that need to be honored; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security

- **External interfaces required:**
  - Examples of external interfaces are - hardware, software and communication interfaces, user interfaces, report formats, etc.
  - To specify the user interfaces, each interface between the software and the users must be described.
- **One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree Other non-functional requirements:**
  - This section contains a description of non- functional requirements that are neither design constraints nor are external interface requirements.
  - An important example is a performance requirement such as the number of transactions completed per unit time.

Goals of implementation:

- The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed.
- A goal, in contrast to the functional and nonfunctional requirements, is not checked by the customer for conformance at the time of acceptance testing.
- The goals of the implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc.

Organization of the SRS Document:

- The organization of an SRS document is prescribed by the IEEE 830 standard.
- IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it.
- Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged.
- The three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.
- The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics.

Various Sections of SRS:

Introduction

- ***Purpose***: This section should describe where the software would be deployed and how the software would be used.
- ***Project scope:*** This section should briefly describe the overall context within which the software is being developed.
- ***Environmental characteristics:*** This section should briefly outline the environment (hardware and other software) with which the software will interact.

Overall description of organization of SRS document

- ***Product perspective***: This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing system, or it is a new software.

Product features: This section should summarize the major ways in which the software

- would be used.
- ***User classes***: Various user classes that are expected to use this software are identified and described here.
- ***Operating environment***: This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact

- ***Design and implementation constraints:*** In this section, the different constraints on the design and implementation are discussed.
- ***User documentation***: This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting

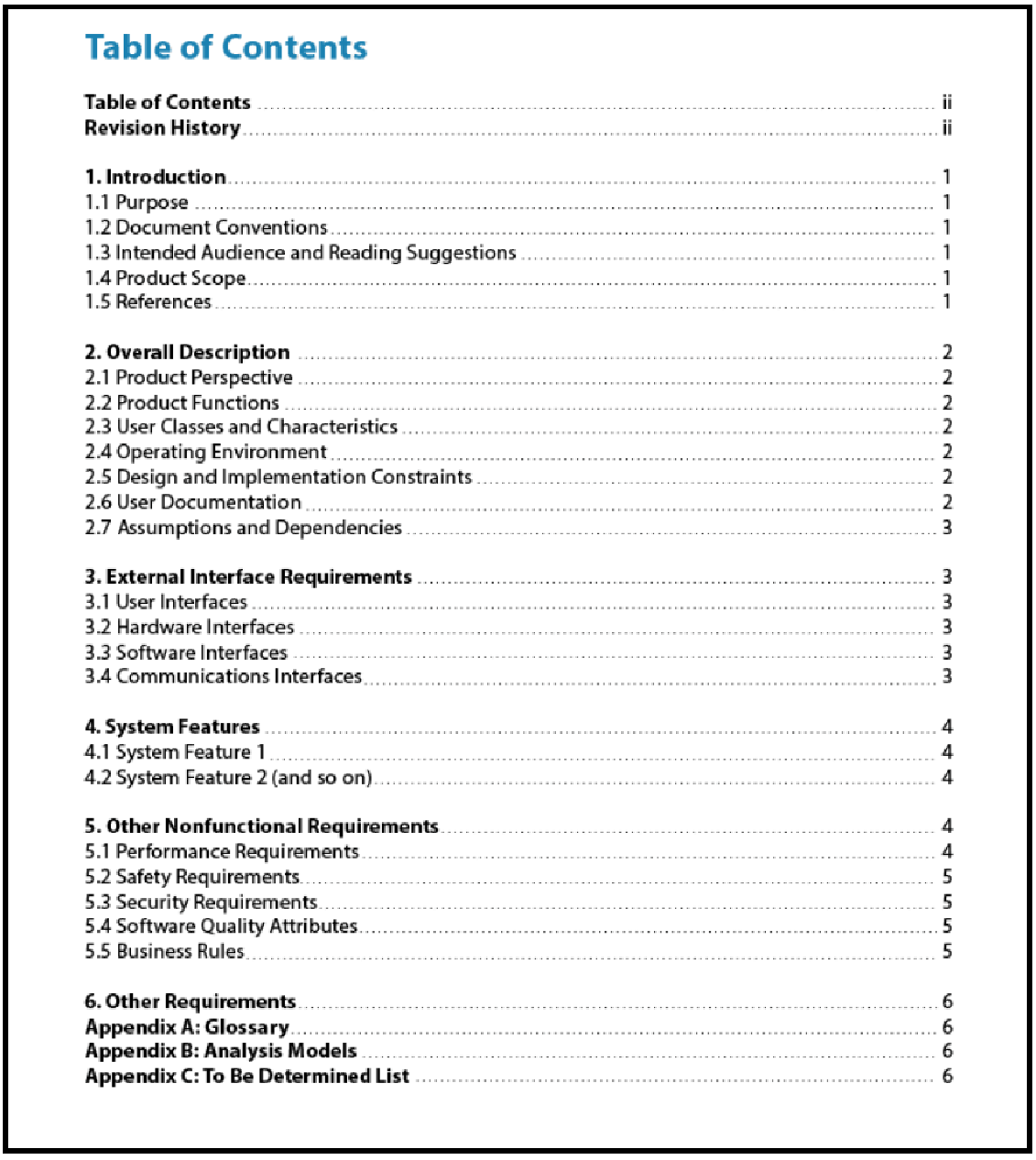


Table of Contents

manuals that will be delivered to the customer along with the software.

IEEE format for SRS Document

External interface requirements

- ***User interfaces:*** This section should describe a high-level description of various interfaces and various principles to be followed.
- ***Hardware interfaces:*** This section should describe the interface between the software and the hardware components of the system

- ***Software interfaces:*** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc.
- ***Communications interfaces:*** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc.

Other non-functional requirements for organization of SRS document

- ***Performance requirements***: Aspects such as number of transactions to be completed per second should be specified here.
- ***Safety requirements:*** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here.
- ***Security requirements:*** This section should specify any requirements regarding security or privacy requirements on data used or created by the software.

IEEE 830 GUDILINES

IEEE 830 is a recommended practice for writing Software Requirements Specifications (SRS), focusing on clear, concise, and complete documentation to facilitate effective communication and development. It provides guidelines for defining what the software should do, how it should interact with other systems, and the constraints it must operate under.

Here's a more detailed breakdown:

- **Purpose:**

IEEE 830 aims to harmonize the content definition for software life cycle process results among IEEE software engineering standards and related international standards.

- **Scope:**

It covers the content and qualities of a good SRS, providing guidelines for specifying requirements for software to be developed, selecting in-house and

commercial software products, and ensuring compliance with relevant standards.

- **Key Aspects:**
- **Clarity and Completeness:** The SRS should be unambiguous, consistent, and complete, leaving no room for misinterpretation.
- **Testability:** Requirements should be written in a way that allows for easy testing and verification.
- **Structure:** The SRS should be well-structured, using a table of contents, introduction, glossary, and sections for functional, non-functional, and interface requirements.
- **Organization:** The SRS should be organized in a way that facilitates easy navigation and understanding.
- **Elements of a good SRS:**
- **Introduction:** Provides context and scope of the software.
- **Functional Requirements:** Describes what the software should do.
- **Non-Functional Requirements:** Describes how the software should perform, such as performance, security, and usability requirements.
- **Interface Requirements:** Describes how the software interacts with other systems.
- **Constraints:** Describes any limitations or restrictions on the software.
- **Superseded by:**

IEEE 830-1998 has been superseded by ISO/IEC/IEEE 29148:2011.

- **Benefits of using IEEE 830:**
- **Improved Communication:** Clear and consistent documentation facilitates better communication between stakeholders.

- **Reduced Errors:** Well-defined requirements can help prevent errors and rework during development.
- **Better Testability:** Testable requirements lead to more thorough testing and higher quality software.
- **Facilitates Selection of Software Products:** The guidelines can be used to select in-house and commercial software products.

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 o shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that the 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.

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.

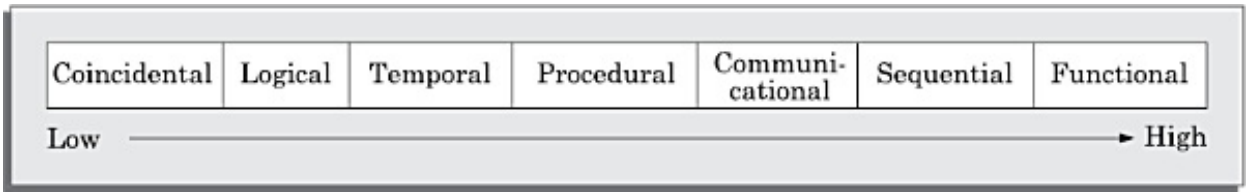


Figure 5.3: Classification of cohesion.

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.

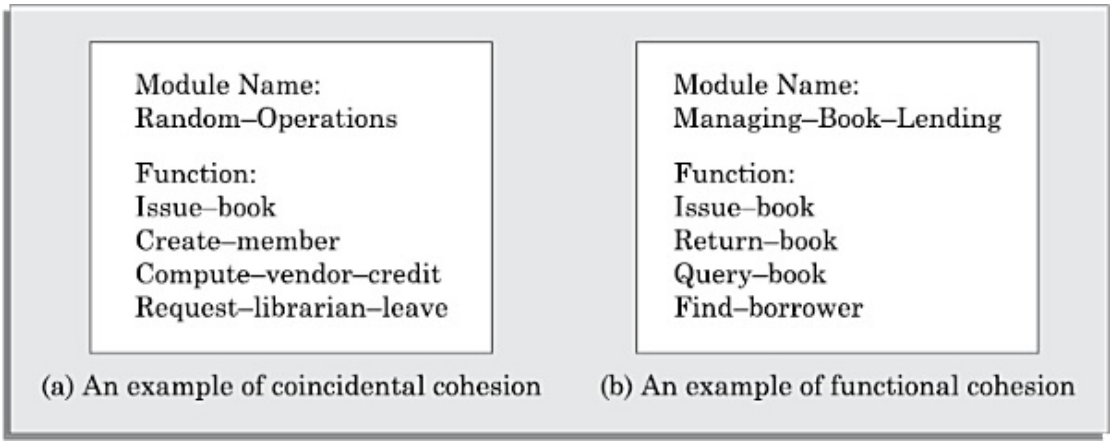


Figure 5.4: Examples of cohesion.

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 admit Student, enter Marks, print Grade Sheet, etc. access and manipulate data stored in an array named student Records 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.,

compute Overtime(), compute Work Hours(), compute Deductions(), etc.) work together to generate the pay slips 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 "initialize", "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.

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.

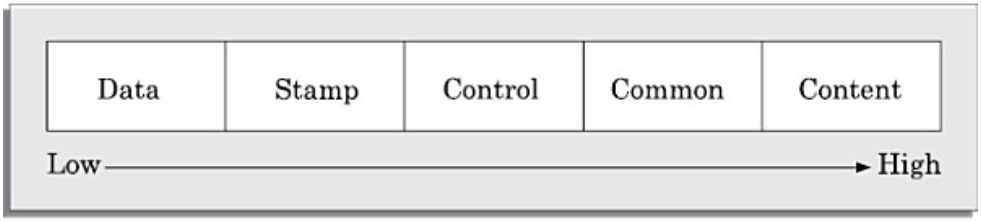


Figure 5.5: Classification of coupling.

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.

LAYERED ARRANGEMENT OF MODULES

The control hierarchy 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:

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

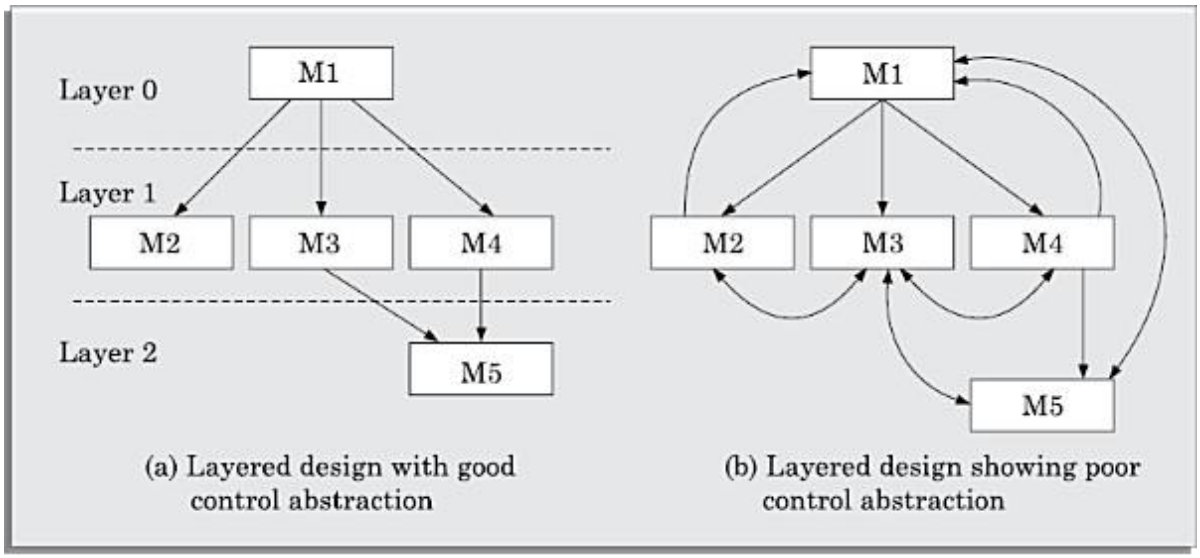
Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

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.



(a) Layered design with good control abstraction

(b) Layered design showing poor control abstraction

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.

Function-oriented Design

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

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

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

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

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

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

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

The system state is centralised and shared among different functions.

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

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

A large number of function-oriented design approaches have been proposed in the past. A

few of the well-established function-oriented design approaches are as following

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

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

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

ttributes: location, status
perations: 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.

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:

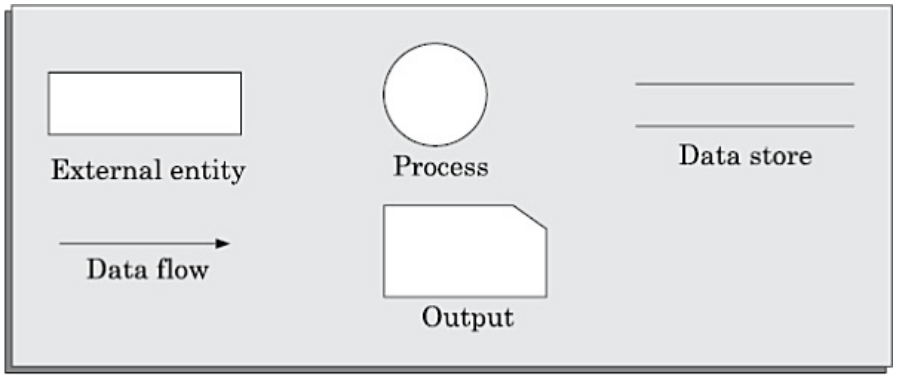


Figure 6.2: Symbols used for designing DFDs.

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

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

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

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

Important concepts associated with constructing DFD models

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

Synchronous and asynchronous operations

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

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

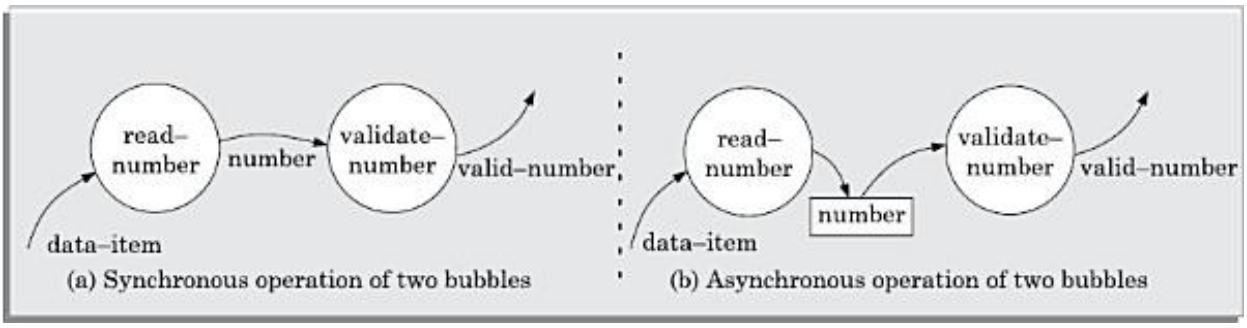


Figure 6.3: Synchronous and asynchronous data flow.

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

6.1 DEVELOPING THE DFD MODEL OF A SYSTEM

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

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

The DFD model of a system i s constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their sub processes and the data flow among these sub processes 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.

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'

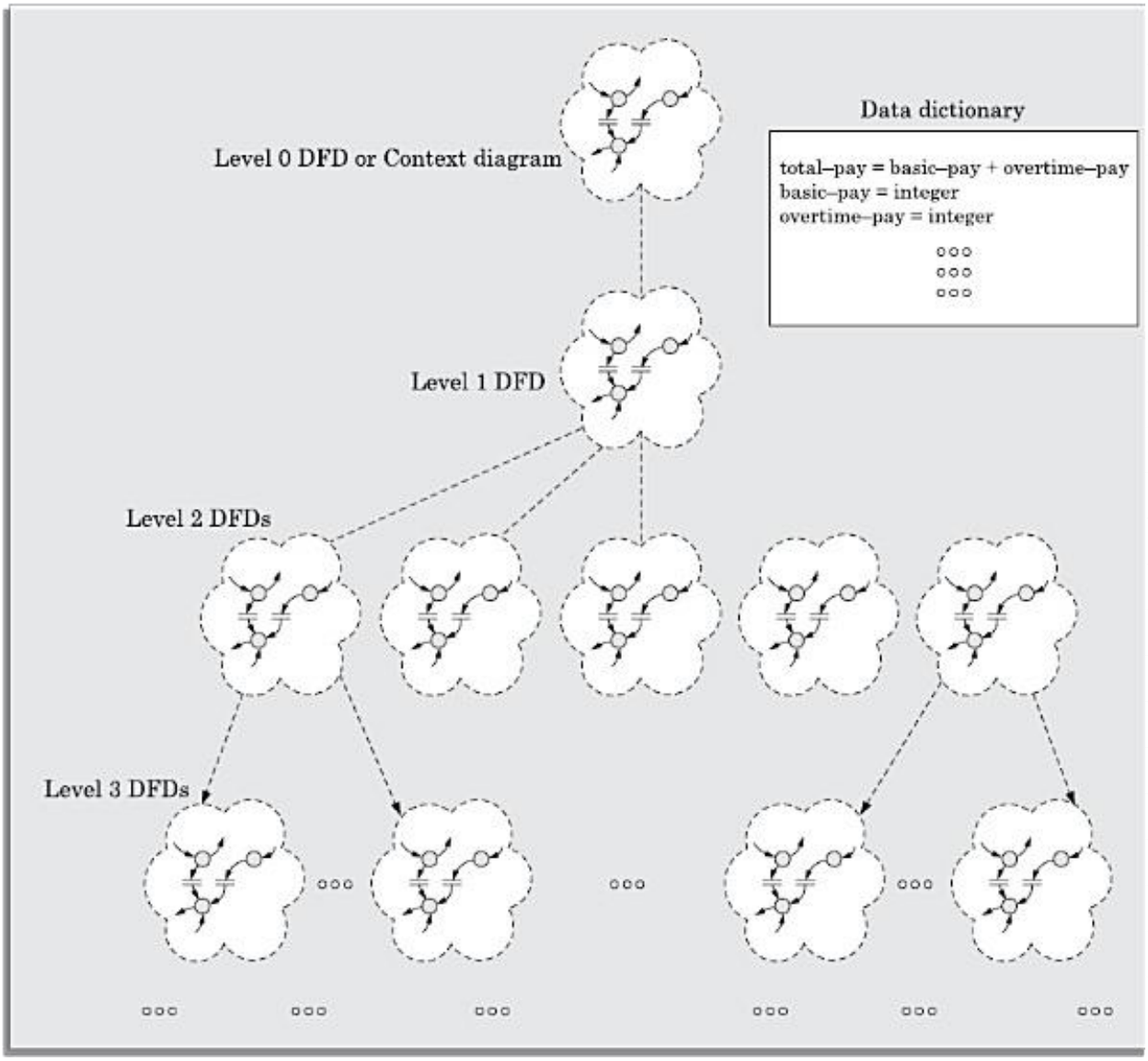


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a singl data dictionary.

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 th context in which the system would exist; that is, the external entities who woul interact with the system and the specific data items that they would be supplying th system and the data items they would be receiving from the system. The variou external entities with which the system interacts and the data flow occurring betwee

the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows These data flow arrows should be annotated with the corresponding data names.

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

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

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

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

1. **Construction of context diagram:** Examine the SRS document to determine:
   - Different high-level functions that the system needs to perform.
   - Data input to every high-level function.
   - Data output from every high-level function.
   - Interactions (data flow) among the identified high-level functions. Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level data flow diagram (DFD), usually called the DFD 0.

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

   Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:
   - Identify the different subfunctions of the high-level function.
   - Identify the data input to each of these subfunctions.
   - Identify the data output from each of these subfunctions.
   - Identify the interactions (data flow) among these subfunctions. Represent these aspects in a diagrammatic form using a DFD. Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing DFDs

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

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

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

How far to decompose?

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

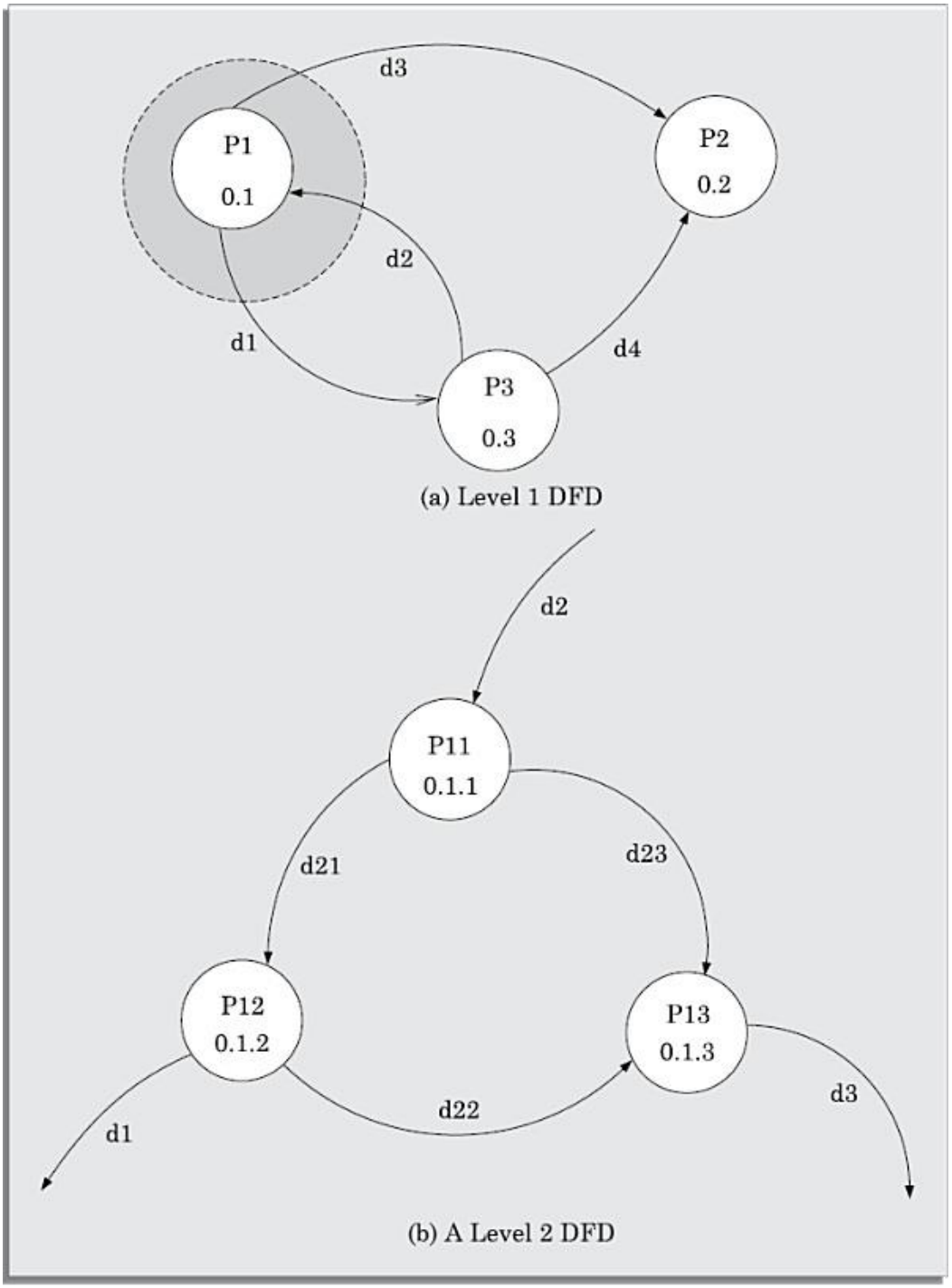


Figure 6.5: An example showing balanced decomposition.

Commonly made errors while constructing a DFD model

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

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

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

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

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

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

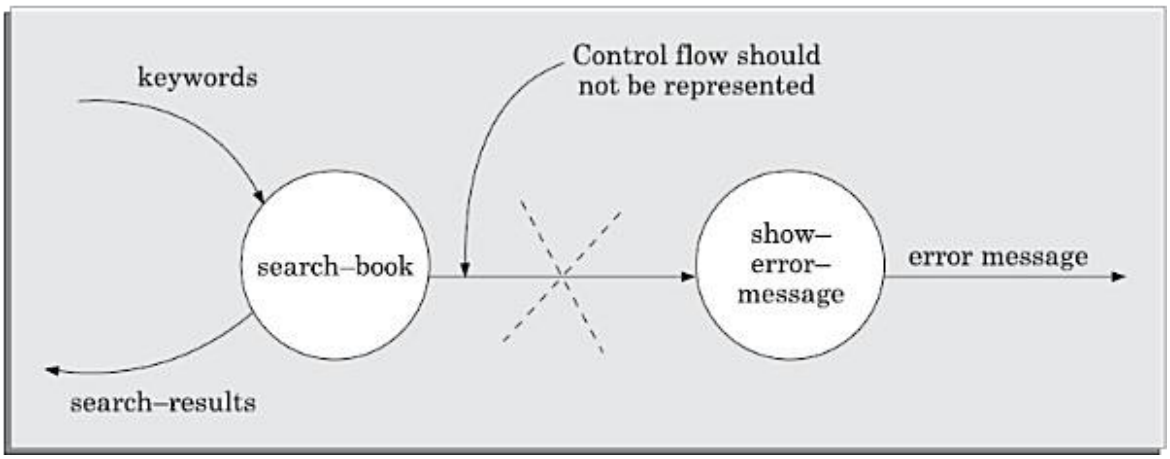


Figure 6.6: It is incorrect to show control information on a DFD.

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

Illustration 3. If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data flow arrow should not connect two data stores or even a data store with an external entity. Thus, data cannot flow from a data store to another data store or to an external entity without any intervening processing. As a result, a data store should be connected only to bubbles through data flow arrows.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in the SRS document of the system should be overlooked.
- Only those functions of the system specified in the SRS document should be represented. That is, the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
- Incomplete data dictionary and data dictionary showing incorrect composition of data items are other frequently committed mistakes.
- The data and function names must be intuitive. Some students and even practicing developers use meaningless symbolic data names such as a,b,c, etc. Such names hinder understanding the DFD model.

- Novices usually clutter their DFDs with too many data flow arrow. It becomes difficult to understand a DFD if any bubble is associated with more than seven data flows. When there are too many data flowing in or out of a DFD, it is better to combine these data items into a high- level data item. Figure 6.7 shows an example concerning how a DFD can be simplified by combining several data flows into a single high- level data flow.

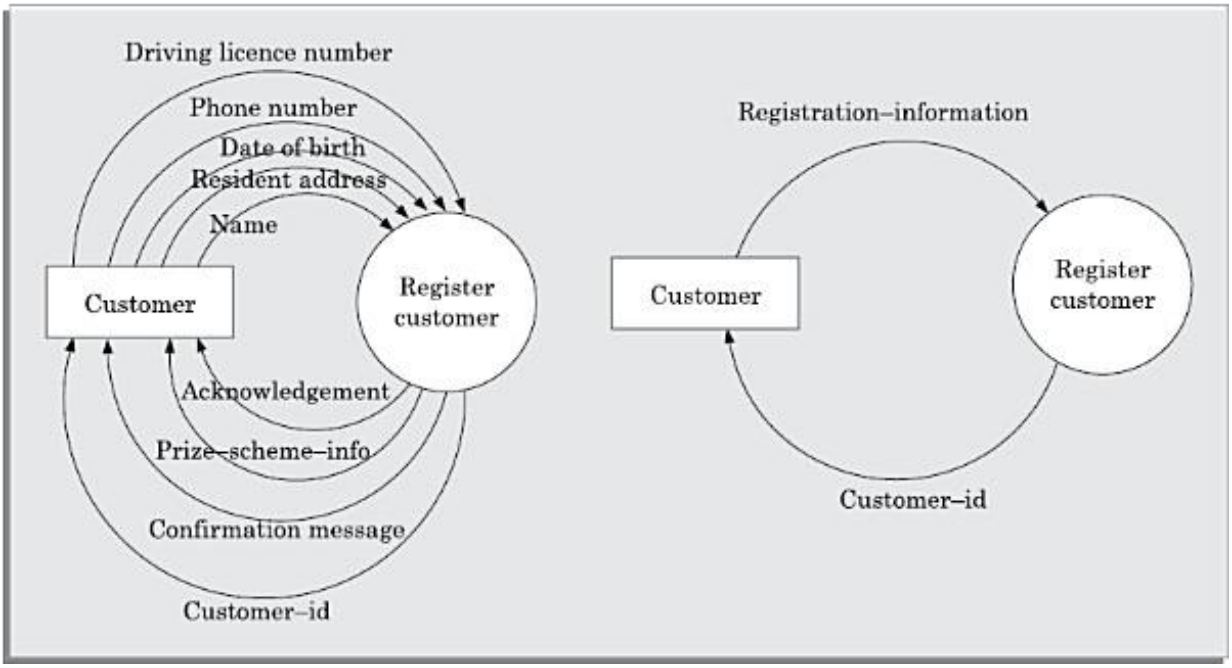


Figure 6.7: Illustration of how to avoid data cluttering.

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 i the level 2 DFD in Figure 6.8(c).

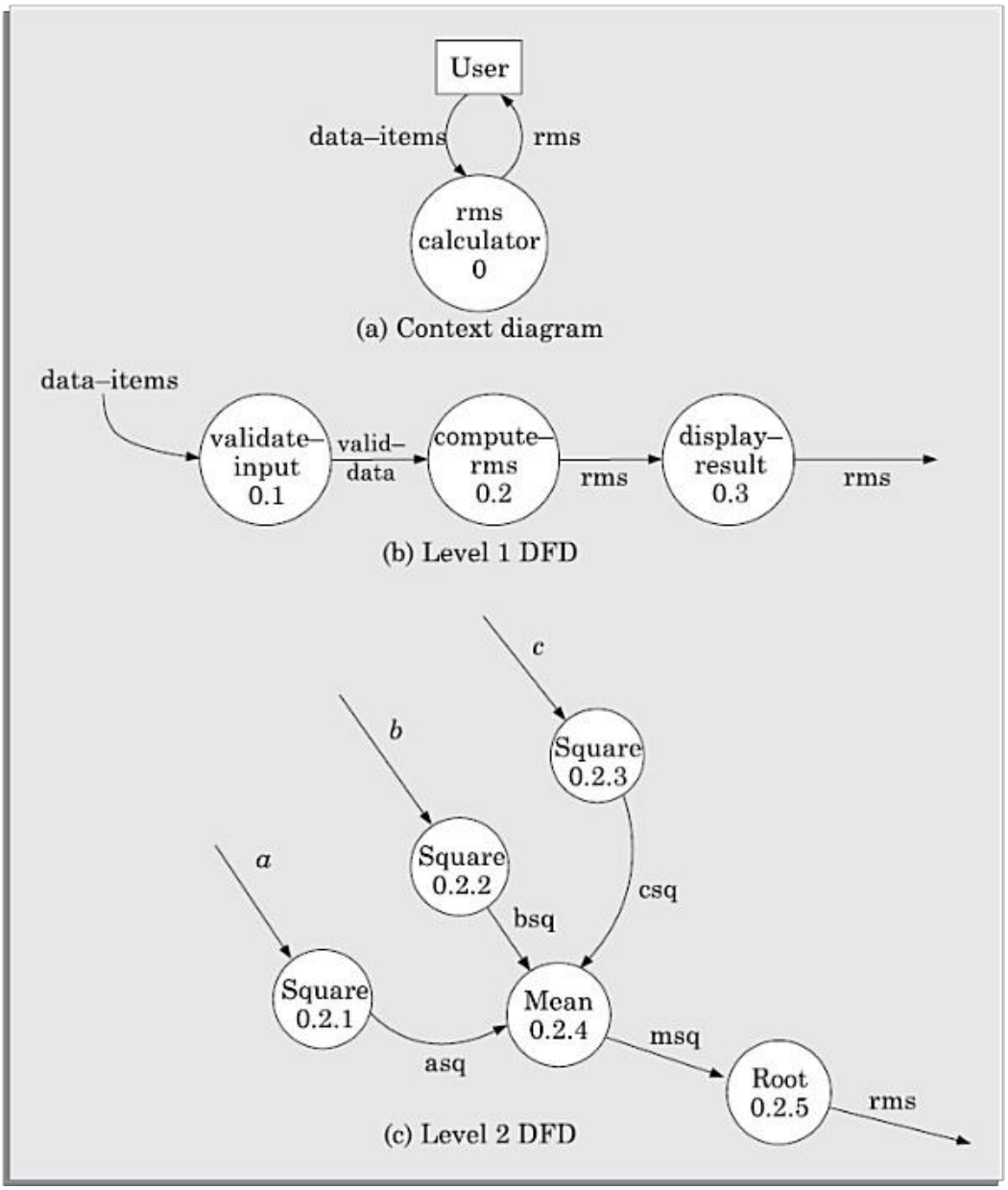


Figure 6.8: Context diagram, level 1, and level 2 DFDs for Example 6.1.

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.

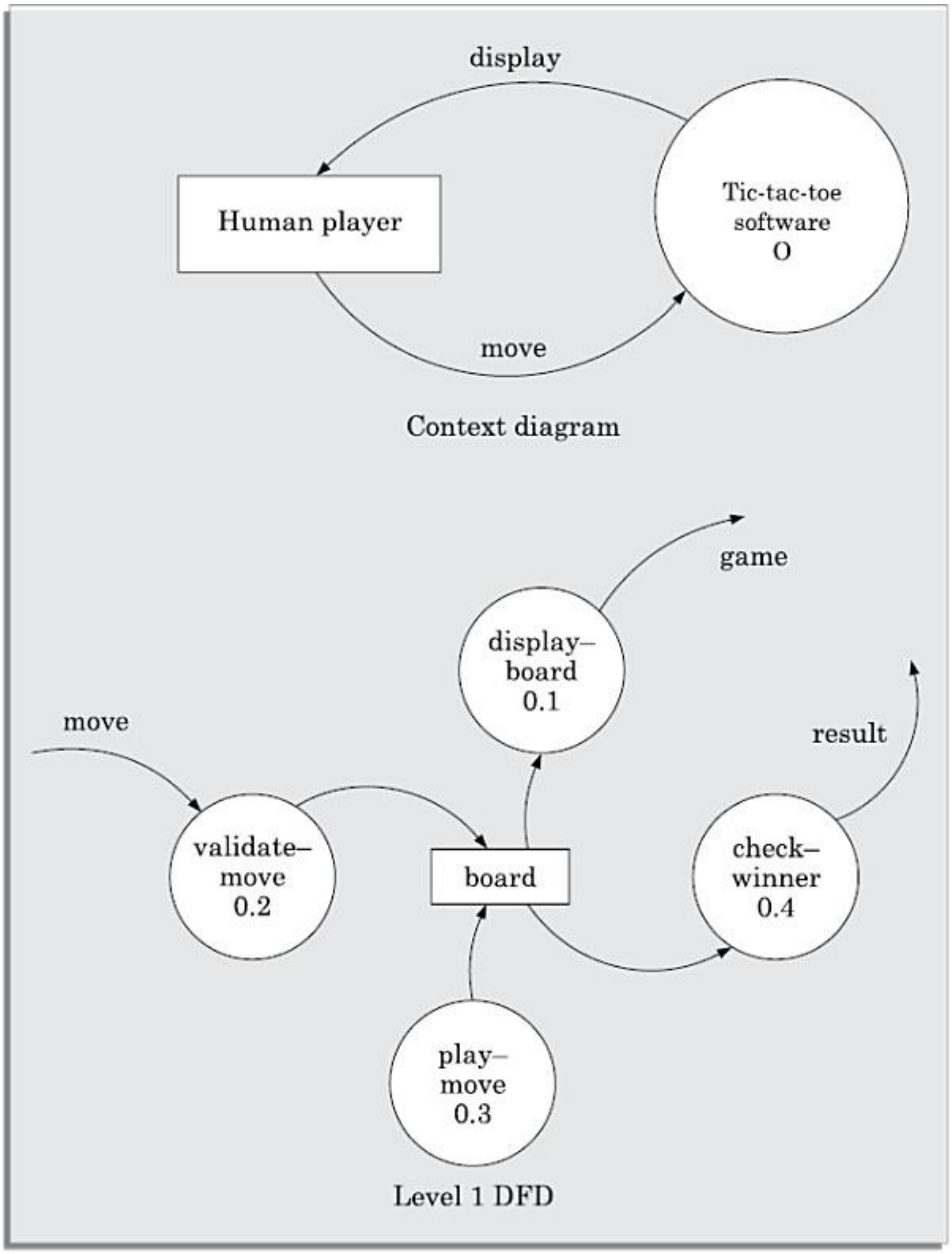


Figure 6.9: Context diagram and level 1 DFDs for Example 6.2.

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.

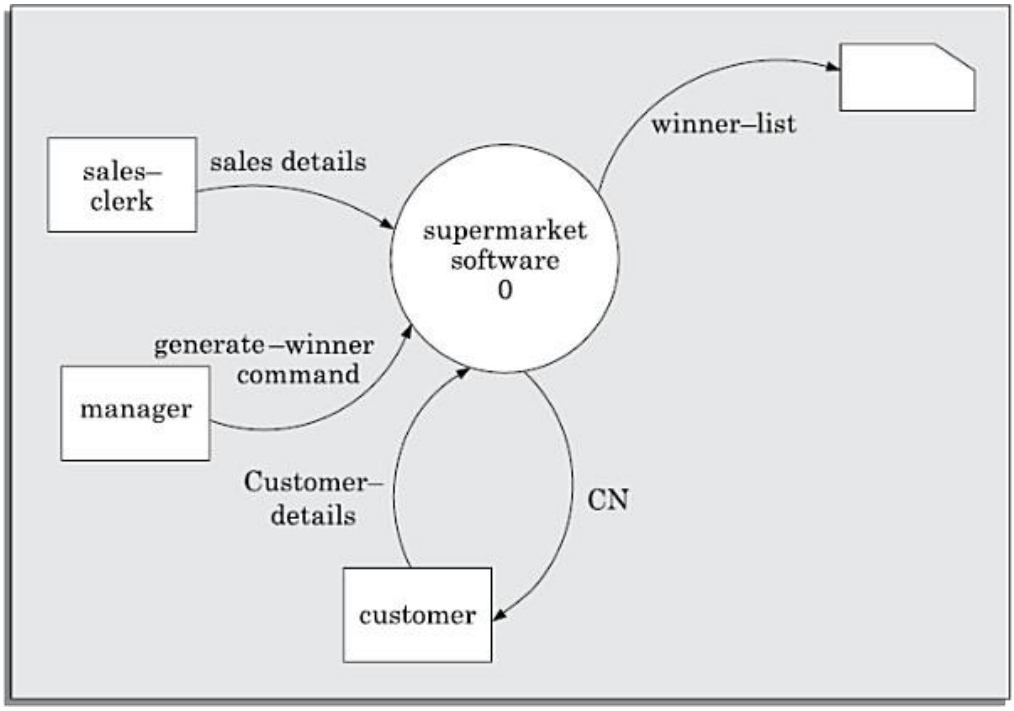


Figure 6.10: Context diagram for Example 6.3.

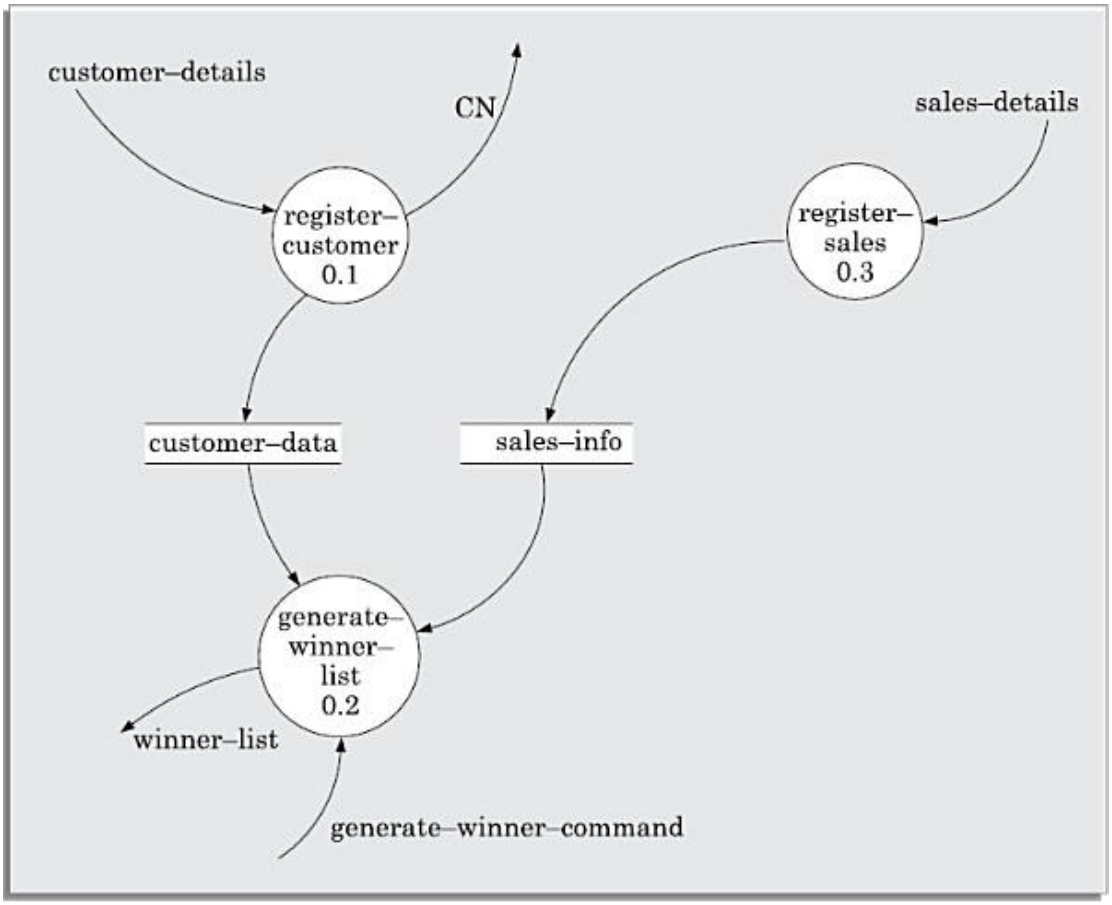


Figure 6.11: Level 1 diagram for Example 6.3.

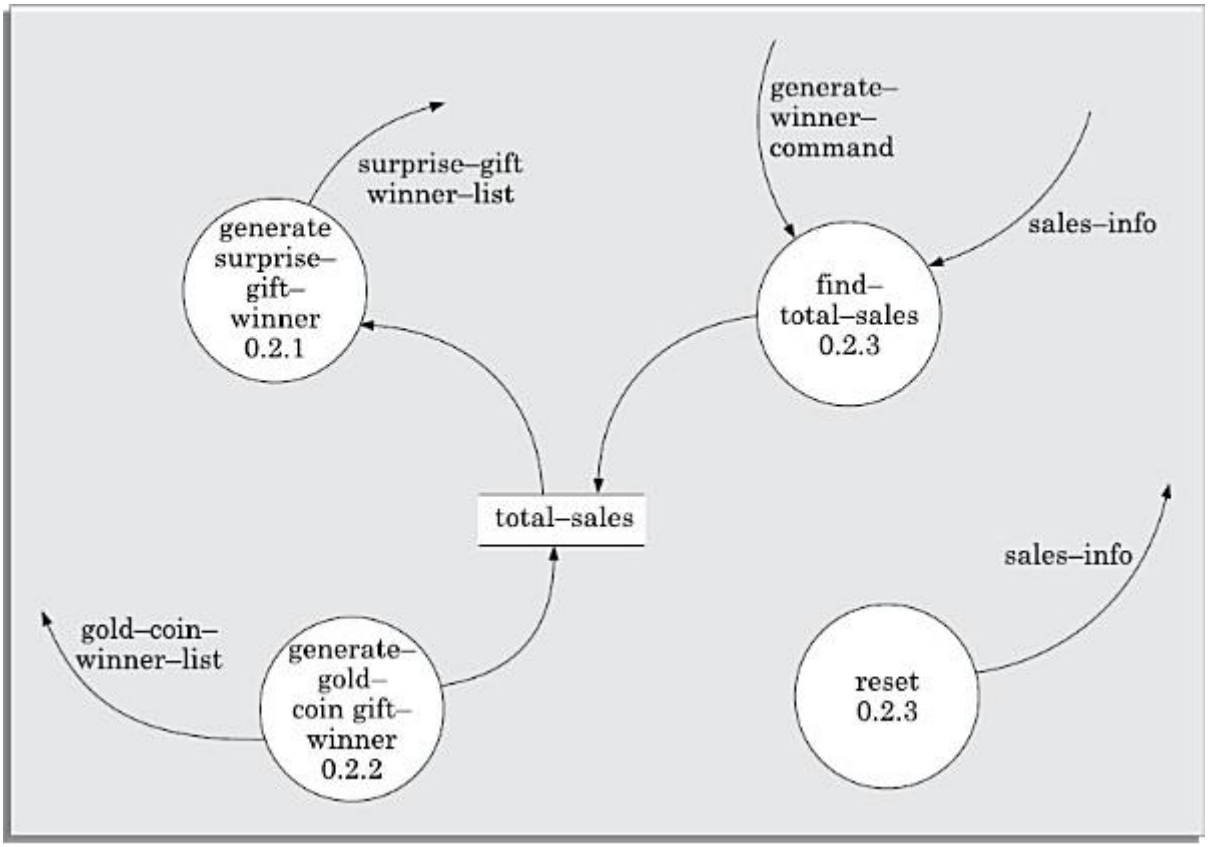


Figure 6.12: Level 2 diagram for Example 6.3.

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) a n d 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 w e 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.

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 interfaceis 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.

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**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.

upport for multiple skill levels: A good user interface should support multiple levels f sophistication of command issue procedure for different categories of users. This is

ıecessary because users with different levels of experience in using an application ›refer different types of user interfaces. Experienced users are more concerned about he efficiency of the command issue procedure, whereas novice users pay importance to ısability aspects. Very cryptic and complex commands discourage a novice, whereas ·laborate command sequences make the command issue procedure very slow and herefore put off experienced users. When someone uses an application for the first ime, his primary concern is speed of learning. After using an application for extended ›eriods of time, he becomes familiar with the operation of the software. As a user ›ecomes more and more familiar with an interface, his focus shifts from usability ·spects to speed of command issue aspects. Experienced users look for options such as hot-keys", "macros", etc.

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.

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.

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.

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.

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.

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 memoris the set of primitive commands. Also, most users make errors while formulatin commands in the command language and also while typing them. Further, in 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 ar 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.

Menu-based Interface

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Humans are much better in recognising something than recollecting it. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu- based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu- based system. Also, if the number of choices is large, it is difficult to design a menu-based interfae. A moderate-sized software might need hundreds or thousands of different menu choices. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms. In the following, we discuss some of the techniques available to structure a large number of menu items:

Scrolling menu: Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up or down to find the size he is looking for. However, if the commands do not have any definite ordering relation, then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organisation inefficient.

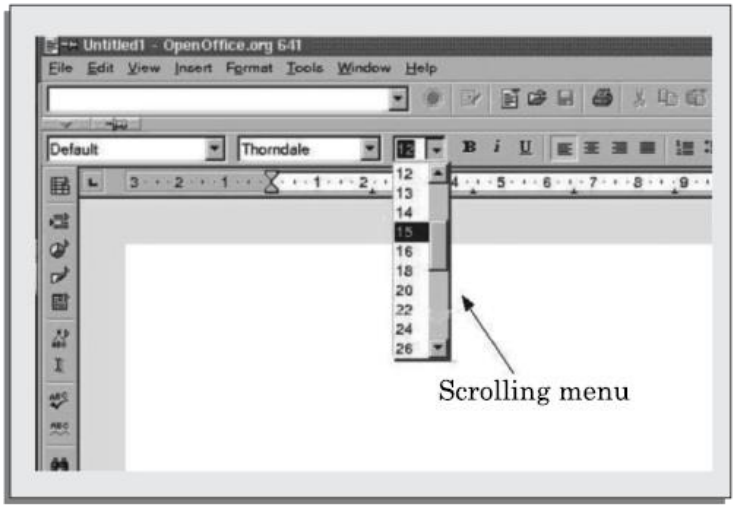


Figure 9.1: Font size selection using scro**l**ing menu.

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.

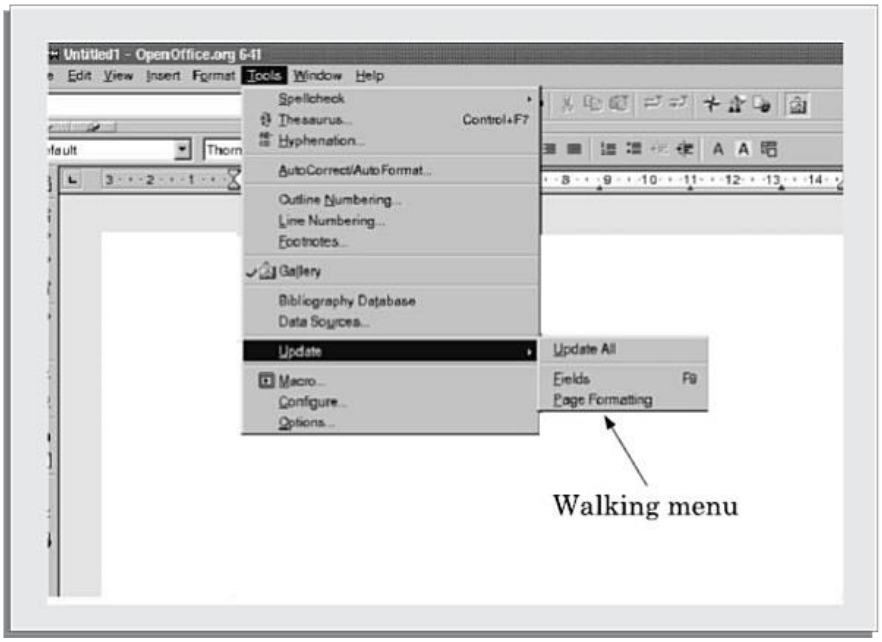


Figure 9.2: Example of walking menu.

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.

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.

Decision Tree and Decision Table

Data is one of the foremost resources of an organization. Every organization wishes to preserve and fully utilize its data for decision-making. Once data is acquired, it must be organized in an application's database for later use.

A database is a collection of facts, rules, and meta-data. There are different ways t organize a database. A variety of knowledge representations techniques, such a semantic nets, frames scripts, lists, decision trees, decision tables, etc., have bee proposed for use over the years. One data representation scheme may be mor efficient than others depending on the nature and type of problem. Thus, there is need to map data from one representation to another.

This mapping may give a faster response and reduce computation amount. Data in th form of rules is easy to understand and fast to extract and implement. Rules can b constructed from the data in a decision tree and decision table.

The blog's primary focus is to construct rules from the data presented in the form of a decision tree and decision table. A human user can understand and modify a set of rules much more easily than he or she can understand and modify a decision tree or decision table.
During Structured Analysis, various techniques and tools are used for system development. These are:

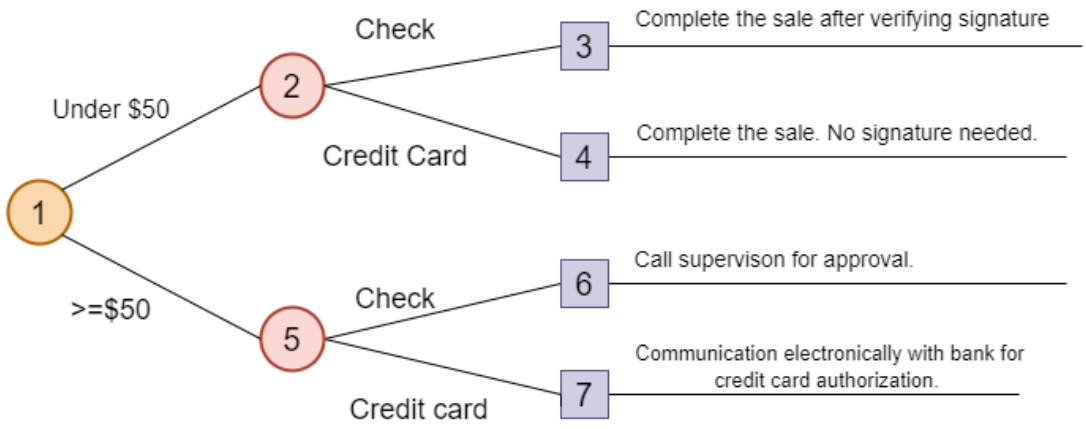
- Data Dictionary

Data Flow Diagrams
Decision Tables
Structured English
Decision Trees
Pseudocode

Decision Tree

A Decision Tree is a graph that uses a branching method to display all the possible outcomes of any decision. It helps in processing logic involved in decision-making, and corresponding actions are taken. It is a diagram that shows conditions and their alternative actions within a horizontal tree framework. It helps the analyst consider the sequence of decisions and identifies the accurate decision that must be made.
Links are used for decisions, while Nodes represent goals. Decision trees simplify the knowledge acquisition process and are more natural than frames and rule knowledge representation techniques.
Let's understand this with an example:
Conditions included the sale amount (under $50) and whether the customer paid by cheque or credit card. The four steps possible were to:
Complete the sale after verifying the signature.
Complete the sale with no signature needed.
Communicate electronically with the bank for credit card authorization.
Call the supervisor for approval.

The below figure illustrates how this example can be drawn as a decision tree. In drawing the tree.



Drawing a decision tree to show the noncash purchase approval actions for a department store.

1. Identify all conditions and actions and their order and timing (if they are critical).
2. Begin building the tree from left to right, making sure you list all possible alternatives before moving to the right.

Advantages of decision trees

Decision trees represent the logic of If-Else in a pictorial form.
Decision trees help the analyst to identify the actual decision to be made.
Decision trees are useful for expressing the logic when the value is variable or actio depending on a nested decision.
It is used to verify the problems that involve a limited number of actions

Also see, V Model in Software Engineering

Decision Tables

Data is stored in the tabular form inside decision tables using rows and columns. A decision table contains condition entries, condition stubs, action entries, and actio stubs. The upper left quadrant contains conditions. The upper right quadrant contain condition alternatives or rules. The lower right quadrant contains action rules, and th lower-left quadrant contains actions to be taken. Verification and validation of th decision table are much easy to check, such as Inconsistencies, Contradictions Incompleteness, and Redundancy.

Example of Decision Table

Let's consider the decision table given in table 1.
In the table, there are multiple rules for a single Decision. The rules from a decision tabl can be made by just putting AND between conditions.
The major rules which can be extracted (taken out) from the table are:
R1 = If (working-day = Y) ^ (holiday = N) ^ (Rainy-day = Y) Then, Go to office.
R2 = If (working-day = N) ^ (holiday = N) ^ (Rainy-day = N) Then, Go to office.
R3 = If (working-day = N) ^ (holiday = Y) ^ (Rainy-day = Y) Then, Watch TV.

- **R4** = If (working-day = N) ^ (holiday = Y) ^ (Rainy-day = N) Then, Go to picnic.

The above rules can be optimized by:
Optimized R1= If (working-day = Y) then Go to office
Or
Optimized R1= If (holiday = N) then Go to office
Optimized R3= If (working-day = N) ^ (Rainy-day = Y) Then Watch TV
Or
Optimized R3= If (holiday = Y) ^ (Rainy-day = Y) Then Watch TV
Optimized R4= If (working-day = N) ^ (Rainy-day = N)
Then go to the picnic.
Or
Optimized R4= If (holiday = Y) ^ (Rainy-day = N)
Then go to the picnic.
The tree given below is the resultant tree of Table 1.
The following rules are constructed from the decision tree as shown below.
R1= If (Day = Working) ^ (Outlook = Rainy)
Then Go To Office

R2= If (Day = Working) ^ (Outlook = Sunny)
Then Go To Office

R3= If (Day = Holiday) ^ (Outlook = Rainy)
Then Watch TV

R4=If (Day = Holiday) ^ (Outlook = Sunny)
Then Go To Picnic

In R1 and R2, there is no need to check the condition Outlook = Rainy and Outlook = Sunny if day = working because if the day is working, whether it is a sunny or rainy day, the decision is to Go to the office. The following rules are the optimized version of R1 and R2 above rules.
R1 optimized: If (Day = Working) Then Go To Office
R2 optimized: If (Day = Working) Then Go To Office
The refinement/optimization step result is effective, efficient, and accurate rules.

Conversion of decision table into decision tree

Data can be transformed from a decision table into a tree structure. The decision table can be converted into a decision tree by using the conversion method discussed or some other technique. The resultant tree has two categories: balanced trees and unbalanced trees. The figure shows the input and output of the conversion process.

Advantages of a Decision tree over Decision table

- The decision tree takes advantage of the sequential structure of decision tre branches to notice the order of checking conditions and executing actions immediately Decision tree is used to verify the problems that involve a limited number of actions.

All those actions and conditions that are critical are connected directly to othe conditions and actions, whereas the conditions that do not matter are absent. In othe words, the trees do not have to be symmetrical.

Decision tree is helpful to express the logic when the value is variable, or action i dependent on the nested decision.

MODULE-III

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.

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.

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 Testing i 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 isexplained as follows. Eliminating an error from code involves three main activities—testing,

debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time consuming activity. In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error.

Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review technique are very different. In the following two subsections, we discuss these two cod review techniques.

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.

Code Inspection

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution

carried out during code walkthroughs. We can state the principal aim of the code inspection to be the following:

The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed

to another programmer's errors.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more likely 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.

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.

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.

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 mos 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.

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 □ (23/1) + (4/23) □ 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.

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.

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.

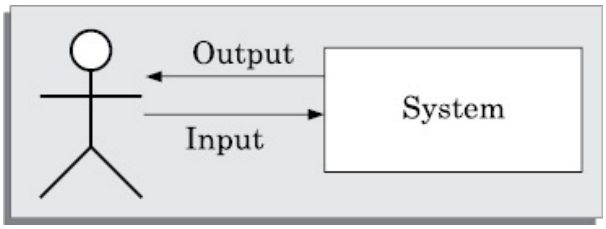


Figure 10.1: A simplified view of program testing.

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.

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.

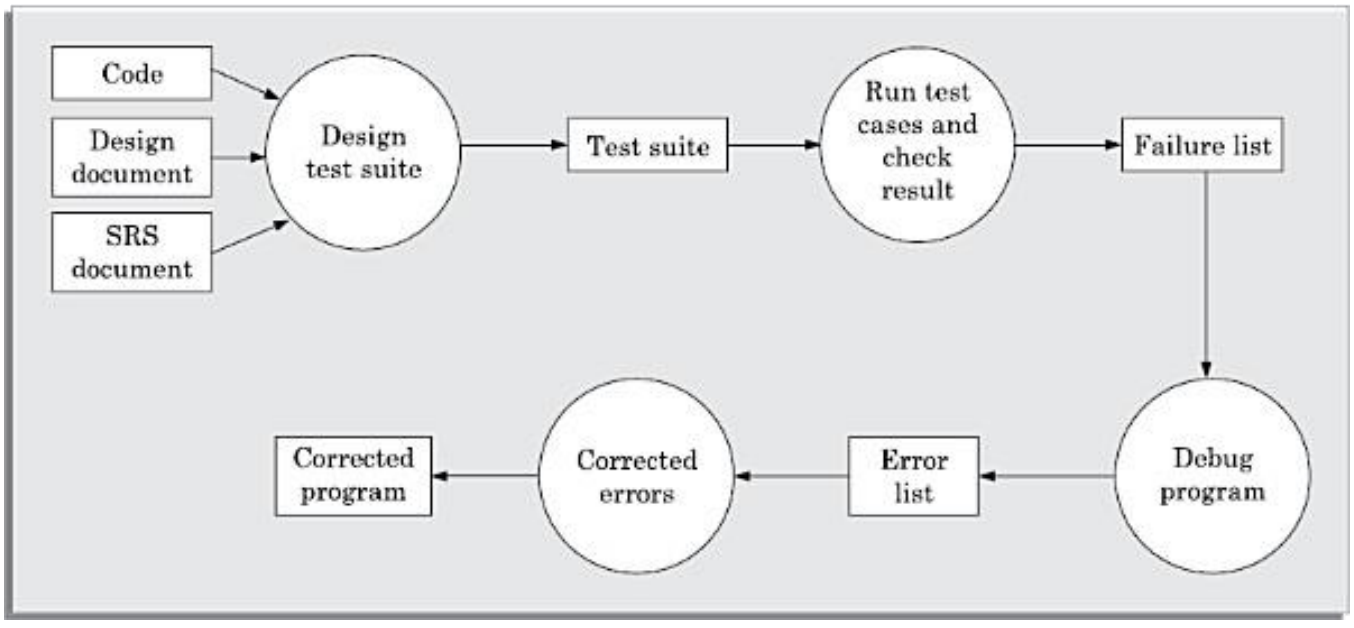


Figure 10.2: Testing process.

10.1.1 Why Design Test Cases?

Before discussing the various test case design techniques, we need to convince ourselve on the following question. Would it not be sufficient to test a software using a larg 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 th 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.

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.

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.

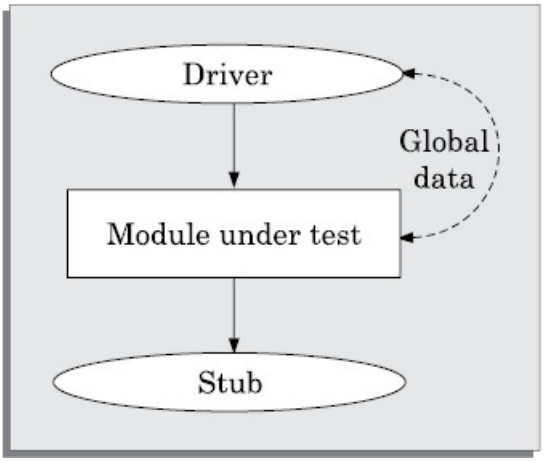


Figure 10.3: Unit testing with the help of driver and stub modules.

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.

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.

Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are [−∞,0], [11,+∞].
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is □-{A,B,C}, where □ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

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 ($m_1 = m_2, c_1 \neq c_2$)
- Intersecting lines ($m_1 \neq m_2$)
- 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}.

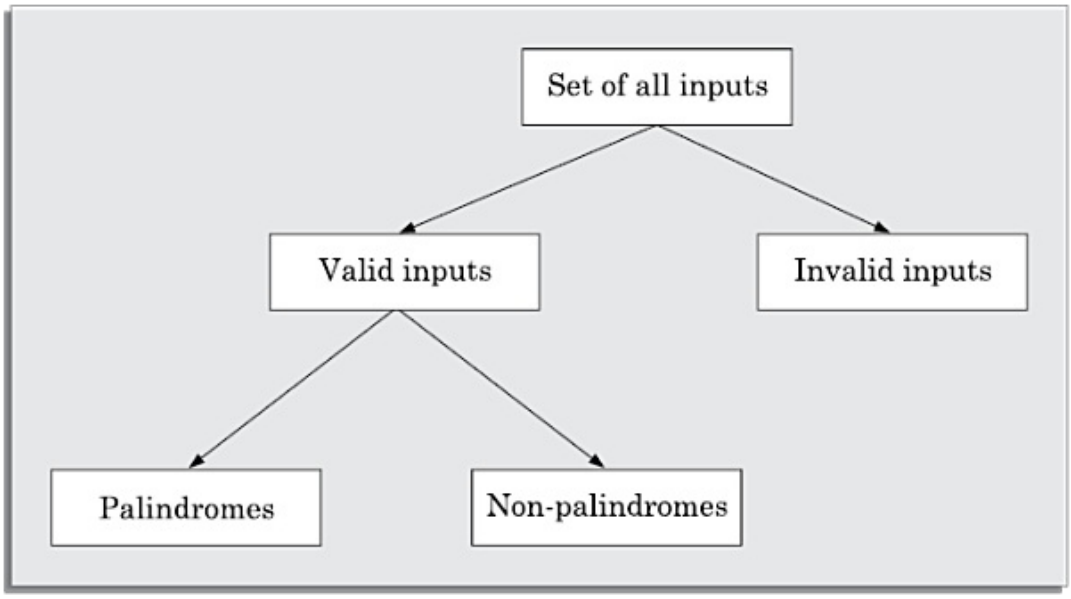


Figure 10.4: Equivalence classes for Example 10.6.

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}.

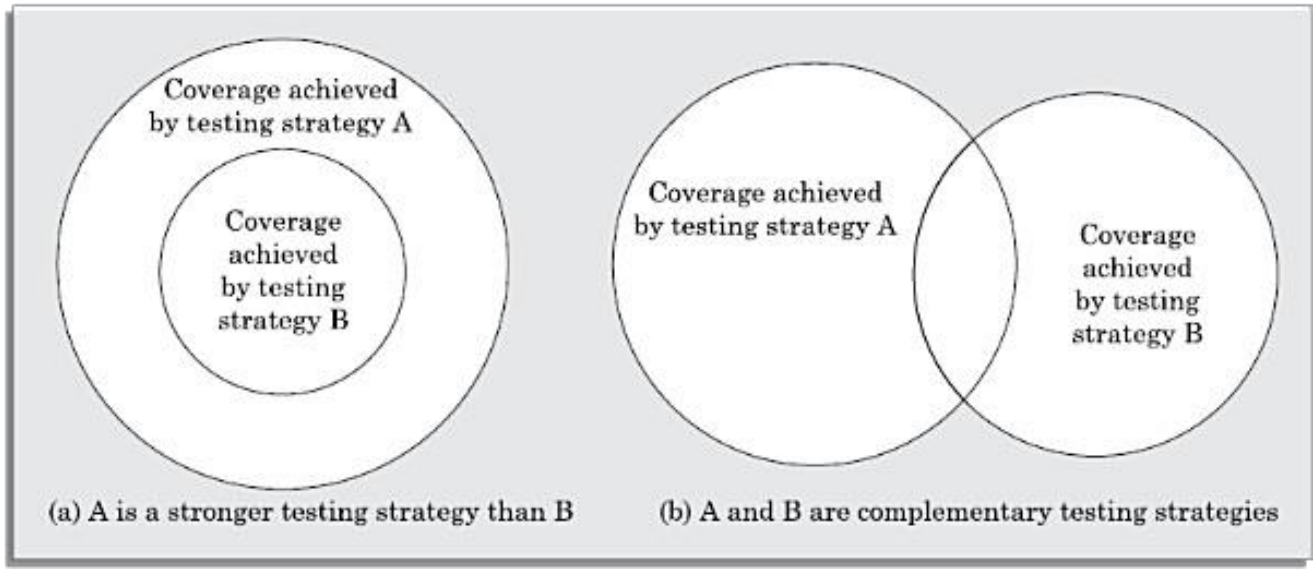


Figure 10.5: CFG for (a) sequence, (b) selection, and (c) iteration type o constructs.

Summary of the Black-box Test Suite Design Approach

We now summarise the important steps in the black-box test suite desig 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.

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.

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.

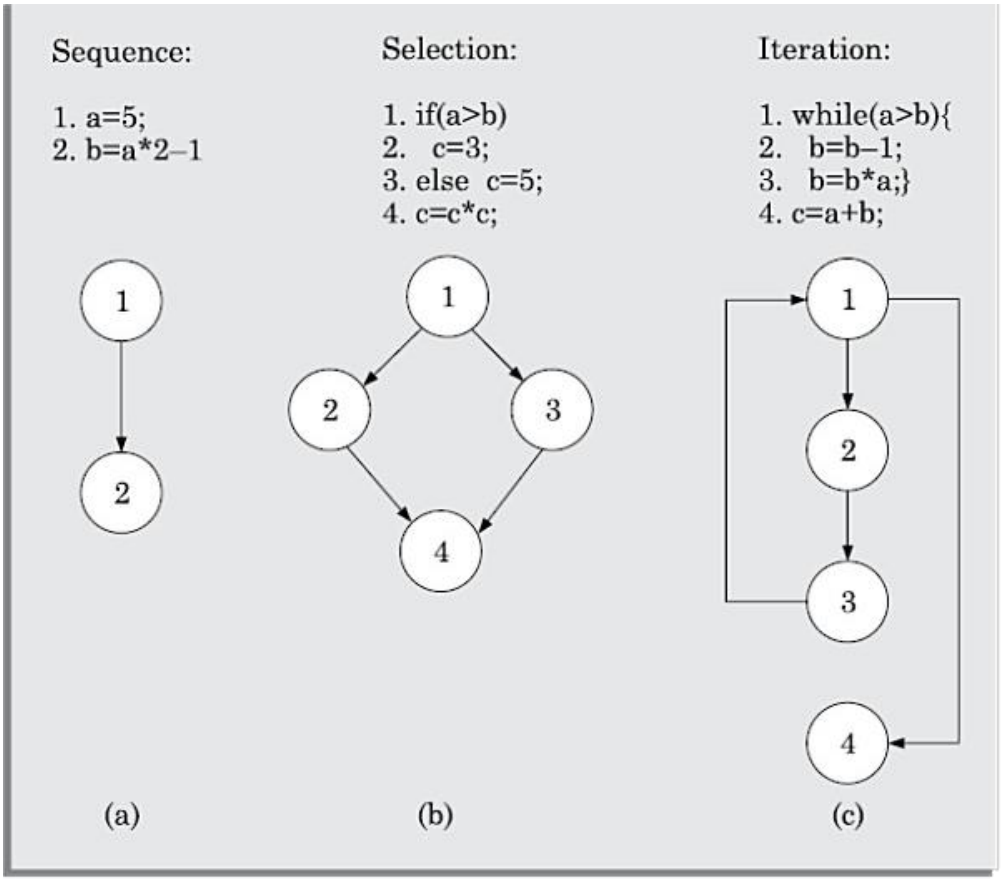


Figure 10.6: I **l**ustration of stronger, weaker, and complementary testing strategies.

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.

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;
{
    1  while (x != y){
    2  if (x>y) then
    3  x=x-y;
    4  else y=y-x;
    5  }
    6  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.

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.

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.

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).

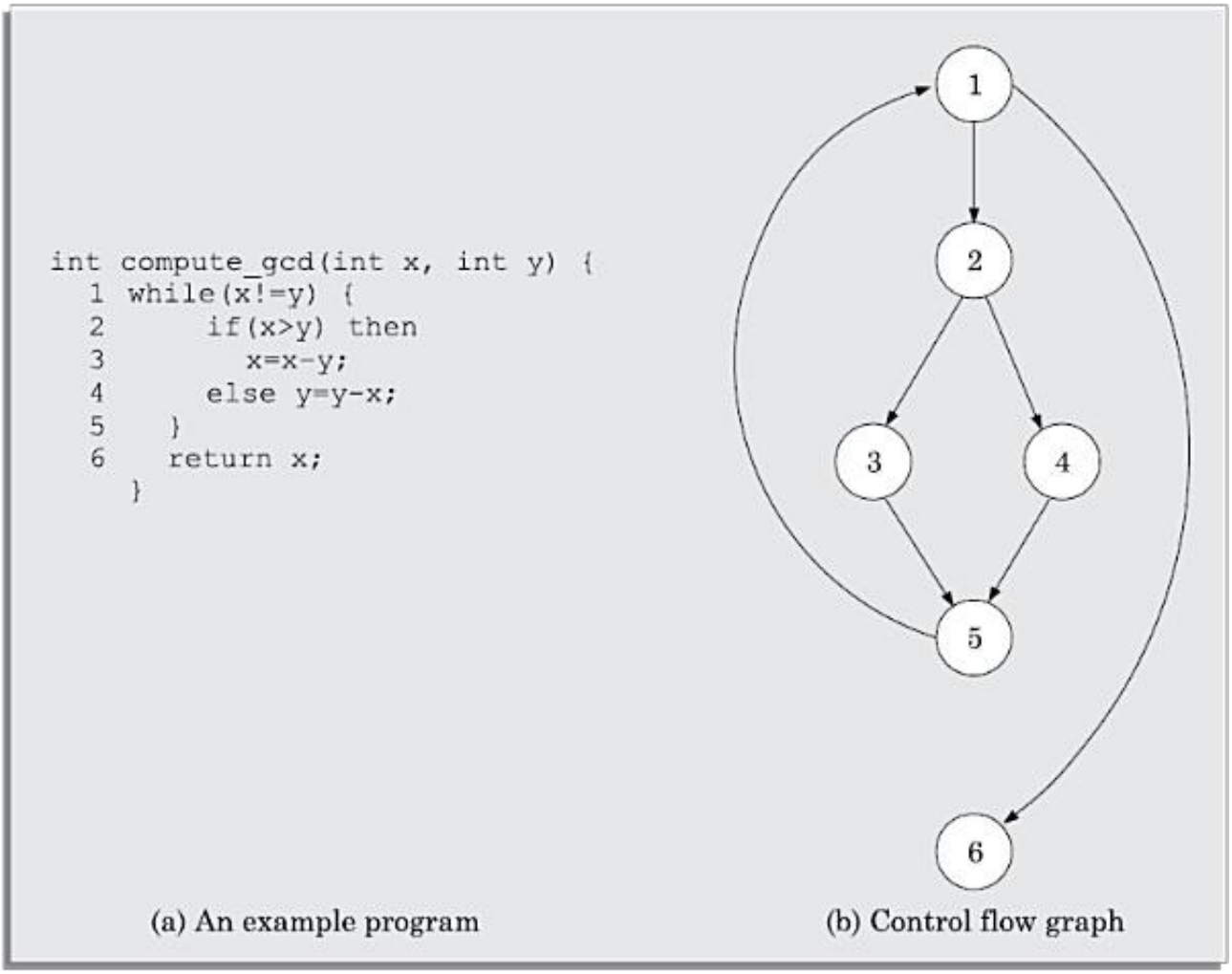


Figure 10.7: Control flow diagram of an example program.

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.

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 o edges in the control flow graph.

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

Method 2: An alternate way of computing the cyclomatic complexity of a program i 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 th 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:

3. Draw control flow graph for the program.
4. Determine the McCabe's metric V(G).
5. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
6. 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

Do Various Object-orientation Feature 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.

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.

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 o 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 stat 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.

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.

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 i 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.

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.

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

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

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.

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.

- In this chapter we discussed the coding and testing phases of the software life cycle.
- Most software development organisations formulate their own coding standards and expect their engineers to adhere to them. On the other hand, coding guidelines serve as general suggestions to programmers regarding good programming styles, but the implementation of the guidelines is left to the discretion to the individual engineers.
- Code review is an efficient way of removing errors as compared to testing, because code review identifies errors whereas testing identifies failures. Therefore, after identifying failures, additional efforts (debugging) must be done to locate and fix the errors.
- Exhaustive testing of almost any non-trivial system is impractical. Also, random selection of test cases is inefficient since many test cases become redundant as they detect the same type of errors. Therefore, we need to design an minimal set of test cases that would expose as many errors as possible.
- There are two well-known approaches to testing—black-box testing and white-box testing. Black box testing is also known as functional testing. Designing test cases for black box testing does not require any knowledge about how the functions have been designed and implemented. On the other hand, white-box testing requires knowledge about internals of the software.
- Object-oriented features complicate the testing process as test cases have to be designed to detect bugs that are associated with these new types of features that are specific to object-orientation programs.
- We discussed some important issues in integration and system testing. We observed that the system test suite is designed based on the SRS document. The two major types of system testing are functionality testing and performance testing. The functionality test cases are designed based on the functional requirements and the performance test cases are design to test the compliance of the system to test the non-functional requirements documented in the SRS document.

MODULE-IV

Basic concepts in software reliability

oftware reliability refers to the probability of a program operating without failures for pecified time in a given environment, and is a crucial aspect of software quality, focusin n the dynamic, operational behavior of a program rather than its static design.

Software Reliability Measures

oftware reliability measures, also known as software reliability metrics, are used t uantify the reliability of a software product. These metrics help developers, testers, an takeholders understand how likely the software is to perform its intended function ithout failure. Here are some of the most common and important measures:

asic Measures

- Mean Time To Failure (MTTF):
  - This measures the average time between consecutive failures.
  - It's particularly relevant for systems that are expected to run for extended periods.
  - A higher MTTF indicates better reliability.
  - Formula: MTTF = Total operating time / Number of failures
- Mean Time To Repair (MTTR):
  - This measures the average time it takes to repair a software system after a failure.
  - A lower MTTR indicates better maintainability and, indirectly, better reliability (as the system is down for a shorter time).
- Mean Time Between Failures (MTBF):
  - This is the average time between two successive failures.
  - For repairable systems, it's the sum of MTTF and MTTR.
  - Formula: MTBF = MTTF + MTTR
- Rate of Occurrence of Failure (ROCOF):
  - This measures the frequency of failures in a given time interval.
  - It's calculated as the number of failures divided by the time of exposure.
- Probability of Failure on Demand (POFOD):
  - This measures the likelihood that the system will fail when a service request is made.
  - It's useful for systems where services are requested intermittently.

- Formula: POFOD = Number of failures / Number of requests
- Availability:
  - This measures the degree to which a system is operational and accessible when required.
  - It's often expressed as a percentage.
  - Availability is influenced by both MTTF and MTTR.
  - Formula: Availability = MTTF / (MTTF + MTTR)

Reliability Growth Models – Software Engineering

The reliability growth group of models measures and predicts the improvement o eliability programs through the testing process. The growth model represents th eliability or failure rate of a system as a function of time or the number of test cases Models included in this group are as follows.

1. Coutinho Model – Coutinho adapted the Duane growth model to represent the software testing process. Coutinho plotted the cumulative number of deficiencies discovered and the number of correction actions made vs. the cumulative testing weeks on log-log paper. Let N(t) denote the cumulative number of failures and let t be the total testing time. The failure rate, λ λ (t), the model can be expressed as[Tex]$$\lambda (t)=\frac{N(t)}{t} $$ $$ =\beta_0t^{-\beta_1}$$ [/Tex]where β0andβ1 $\beta 0 and \beta 1$ are the model parameters. The least squares method can be used to estimate the parameters of this model.

2. Wall and Ferguson Model – Wall and Ferguson proposed a model similar to the Weibull growth model for predicting the failure rate of software during testing. The cumulative number of failures at time t, m(t), can be expressed as[Tex]$$m(t)=a_0[b(t)]^\beta $$ [/Tex]where α0andα1 $\alpha 0 and \alpha 1$ are the unknown parameters. The function b(t) can be obtained as the number of test cases or total testing time. Similarly, the failure rate function at time t is given by [Tex]$$\lambda (t)= {m^’ (t)} = {a_0\beta b^’ (t){[b(t)]^{\beta -1}}}$$ [/Tex]Wall and Ferguson tested this model using some software failure data and observed that failure data correlate well with the model

Reliability growth models are mathematical models used to predict the reliabilit of a system over time. They are commonly used in software engineering t predict the reliability of software systems and to guide the testing an improvement process.

Types of reliability growth models:

1. Non-homogeneous Poisson Process (NHPP) Model: This model is based on the assumption that the number of failures in a system follows a Poisson distribution. It is used to model the reliability growth of a system over time and to predict the

number of failures that will occur in the future.

2. Duane Model: This model is based on the assumption that the rate of failure of a system decreases over time as the system is improved. It is used to model the reliability growth of a system over time and to predict the reliability of the system at any given time.

3. Gooitzen Model: This model is based on the assumption that the rate of failure of a system decreases over time as the system is improved, but that there may be periods of time where the rate of failure increases. It is used to model the reliability growth of a system over time and to predict the reliability of the system at any given time.

4. Littlewood Model: This model is based on the assumption that the rate of failure of a system decreases over time as the system is improved, but that there may be periods of time where the rate of failure remains constant. It is used to model the reliability growth of a system over time and to predict the reliability of the system at any given time.

5. Reliability growth models are useful tools for software engineers, as they can help to predict the reliability of a system over time and to guide the testing and improvement process. They can also help organizations to make informed decisions about the allocation of resources, and to prioritize improvements to the system.

6. It is important to note that reliability growth models are only predictions, and actual results may differ from the predictions. Factors such as changes in the system, changes in the environment, and unexpected failures can impact the accuracy of the predictions.

Advantages of Reliability Growth Models:

1. Predicting Reliability: Reliability growth models are used to predict the reliability of a system over time, which can help organizations make informed decisions about the allocation of resources and the prioritization of improvements to the system.

2. Guiding the Testing Process: Reliability growth models can be used to guide the testing process, by helping organizations determine which tests should be run, and when they should be run, in order to maximize the improvement of the system's reliability.

3. Improving the Allocation of Resources: Reliability growth models can help organizations to make informed decisions about the allocation of resources, by providing an estimate of the expected reliability of the system over time, and by

helping to prioritize improvements to the system.

4. Identifying Problem Areas: Reliability growth models can help organizations to identify problem areas in the system, and to focus their efforts on improving these areas in order to improve the overall reliability of the system.

Disadvantages of Reliability Growth Models:

1. Predictive Accuracy: Reliability growth models are only predictions, and actual results may differ from the predictions. Factors such as changes in the system, changes in the environment, and unexpected failures can impact the accuracy of the predictions.
2. Model Complexity: Reliability growth models can be complex, and may require a high level of technical expertise to understand and use effectively.
3. Data Availability: Reliability growth models require data on the system's reliability, which may not be available or may be difficult to obtain.

SOFTWARE MAINTENANCE

Many students and practising engineers have a preconceived bias against software maintenanc e work. The mention of the word maintenance brings up the image of a crew driver, wielding mechanic with soiled hands holding onto a bagful of spare parts. t would be the objective of this chapter to clear up this misnomer, provide some ntuitive understanding of the software maintenance projects, and to familiarise you vith the latest techniques in software maintenance.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use. On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

In Section 13.1, we examine some general issues concerning maintenance projects. In Section 13.2, we discuss some basic ideas about software reverse engineering. In Section 13.3 we discuss two software maintenance process models which attempt to systematise the software development effort and finally we discuss some concepts involved in cost estimation of maintenance efforts.

13.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

In this section, we first classify the different maintenance efforts into a few classes. Next, we discuss some general characteristics of the maintenance projects. We also discuss some special problems associated with maintenance projects.

Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the usercommunity to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts.

Types of Software Maintenance

There are three types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Characteristics of Software Evolution

Lehman and Belady have studied the characteristics of evolution of several software products [1980]. They have expressed their observations in the form of laws. Their important laws are presented in the following subsection. But a word of caution here is that these are generalizations and may not be applicable to specific cases and also most of these observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law: A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts. This law clearly shows that every product irrespective of how well designed must undergo maintenance. In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need

maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law: The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that when you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support. If you do not redesign the system, the additions will be more complex that they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law: Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

Special Problems Associated with Software Maintenance

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:
Software maintenance work in organizations is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester year's age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work, and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of softwar products needing maintenance are legacy products. Though the word legacy implie "aged" software, but there is no agreement on what exactly is a legacy system. It i prudent to define a legacy system as any software system that is hard to maintain. Th typical problems associated with legacy systems are poor documentation, unstructure (spaghetti code with ugly control structure), and lack of personnel knowledgeable in th product. Many of the legacy systems were developed long time back. But, it is possibl that a recently developed system having poor design and documentation can b considered to be a legacy system

SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1. A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important because we had seen in Chapter 9 that meaningful variable names are the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

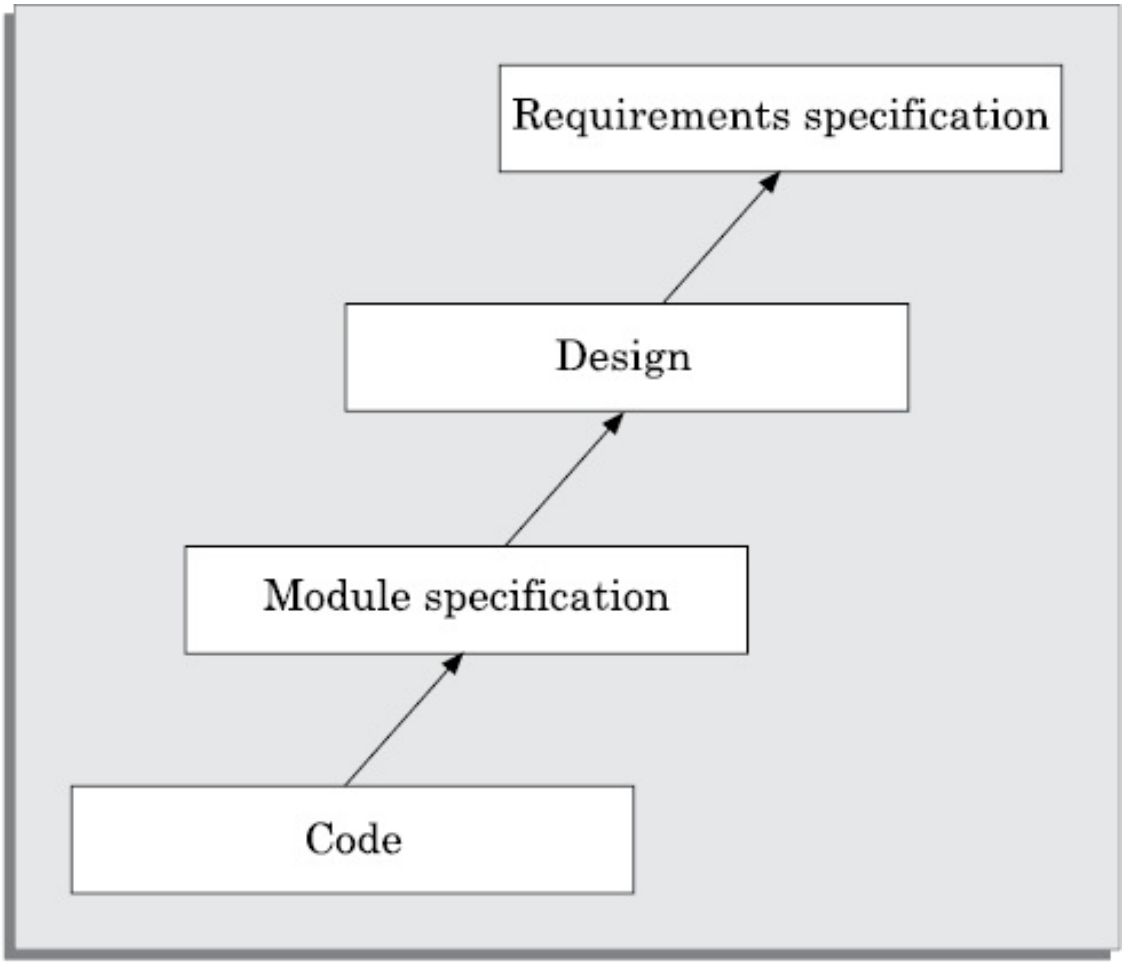


Figure 13.1: A process model for reverse engineering.

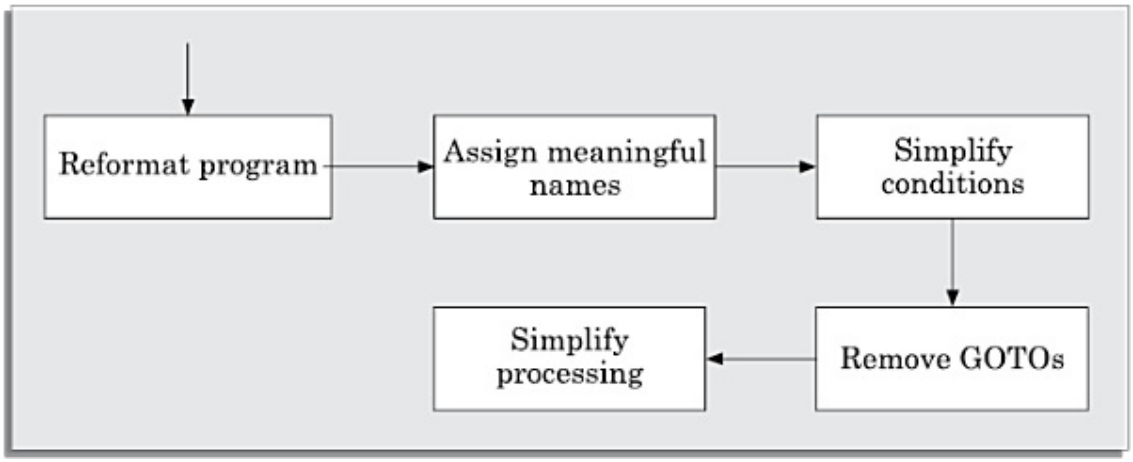


Figure 13.2: Cosmetic changes carried out before reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the proces of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

SOFTWARE MAINTENANCE PROCESS MODELS

Before discussing process models for software maintenance, we need to analyse various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as: (i) the extent of modification to the product required, (ii) the resources available to the maintenance team, (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.), (iii) the expected project risks, etc. When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope (activities required) for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed.

First model

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in Figure 13.3. In this approach, the

project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the re- engineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

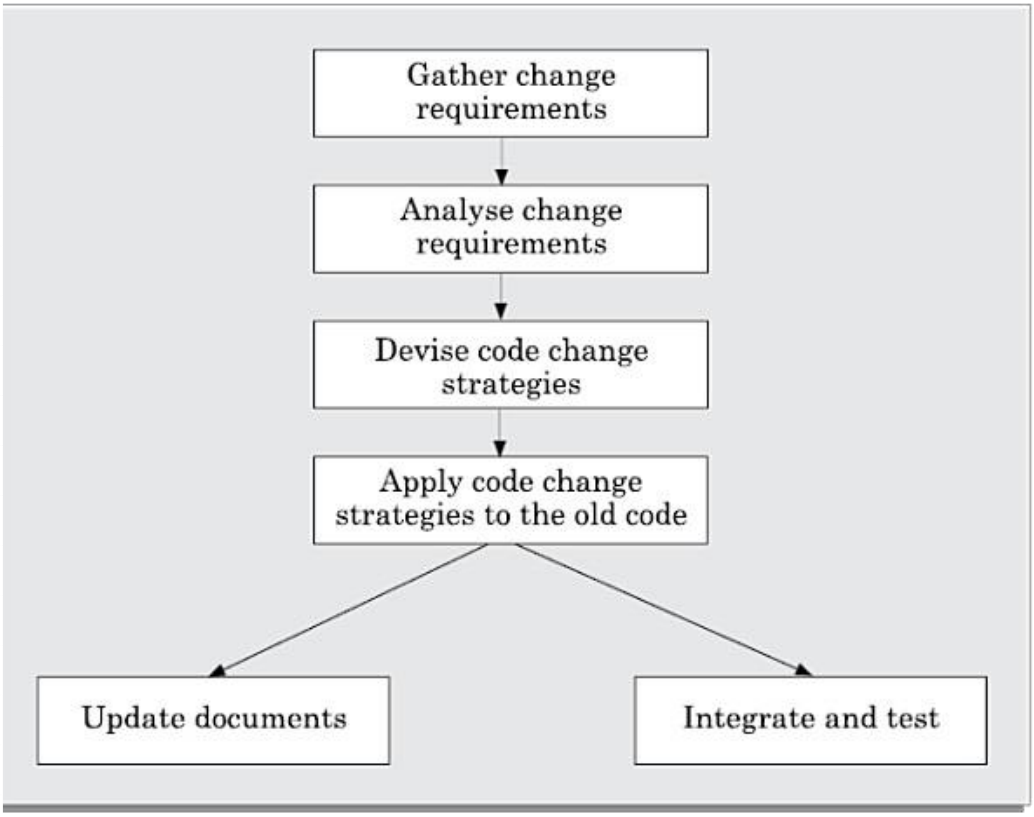
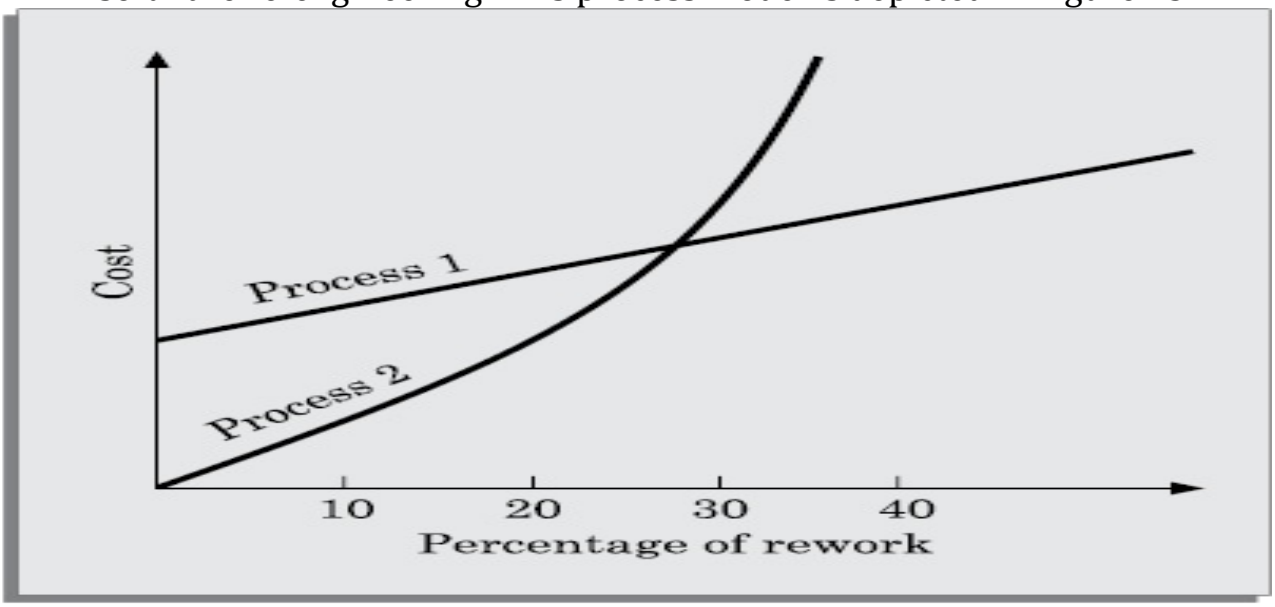


Figure 13.3: Maintenance process model 1.

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering. This process model is depicted in Figure 13.4.



\ **Figure 13.5:** Empirical estimation of maintenance cost versus percentage rework.

Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

- Re-engineering might be preferable for products which exhibit a high failure rate.
- Re-engineering might also be preferable for legacy products having poor design and code structure.

ESTIMATION OF MAINTENANCE COST

We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the annual change traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change

during a typical year either through addition or deletion.

$$\mathrm{ACT} = \frac{\mathrm{KLOC}_{\mathrm{added}} + \mathrm{KLOC}_{\mathrm{deleted}}}{\mathrm{KLOC}_{\mathrm{total}}}$$

Where, KLOCadded is the total kilo lines of source code added during maintenance. KLOCdeleted is the total KLOC deleted during

Maintenance. Thus, the code that is changed, should be counted in both the code added and code deleted.

The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = ACT × Development cost

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

SOFTWARE REENGINEERING

Re-engineering, also known as software re-engineering, is the process of analyzing designing, and modifying existing software systems to improve their quality performance, and maintainability.

1. This can include updating the software to work with new hardware or softwar platforms, adding new features, or improving the software's overall design an architecture.
2. Software re-engineering, also known as software restructuring or softwar renovation, refers to the process of improving or upgrading existing softwar systems to improve their quality, maintainability, or functionality.
3. It involves reusing the existing software artifacts, such as code, design, an documentation, and transforming them to meet new or updated requirements.

Objective of Re-engineering

The primary goal of software re-engineering is to improve the quality an maintainability of the software system while minimizing the risks and cost associated with the redevelopment of the system from scratch. Software re engineering can be initiated for various reasons, such as:

1. To describe a cost-effective option for system evolution.
2. To describe the activities involved in the software maintenance process.
3. To distinguish between software and data re-engineering and to explain th problems of data re-engineering.

Overall, software re-engineering can be a cost-effective way to improve the qualit and functionality of existing software systems, while minimizing the risks and cost associated with starting from scratch.

Process of Software Re-engineering

The process of software re-engineering involves the following steps:

Process of Software Re-engineering

1. **Planning:** The first step is to plan the re-engineering process, which involve identifying the reasons for re-engineering, defining the scope, and establishing th goals and objectives of the process.
2. **Analysis:** The next step is to analyze the existing system, including the code documentation, and other artefacts. This involves identifying the system's strength and weaknesses, as well as any issues that need to be addressed.
3. **Design:** Based on the analysis, the next step is to design the new or update software system. This involves identifying the changes that need to be made an developing a plan to implement them.
4. **Implementation:** The next step is to implement the changes by modifying th existing code, adding new features, and updating the documentation and othe artefacts.
5. **Testing:** Once the changes have been implemented, the software system needs to b tested to ensure that it meets the new requirements and specifications.
6. **Deployment:** The final step is to deploy the re-engineered software system an make it available to end-users.

Why Perform Re-engineering?

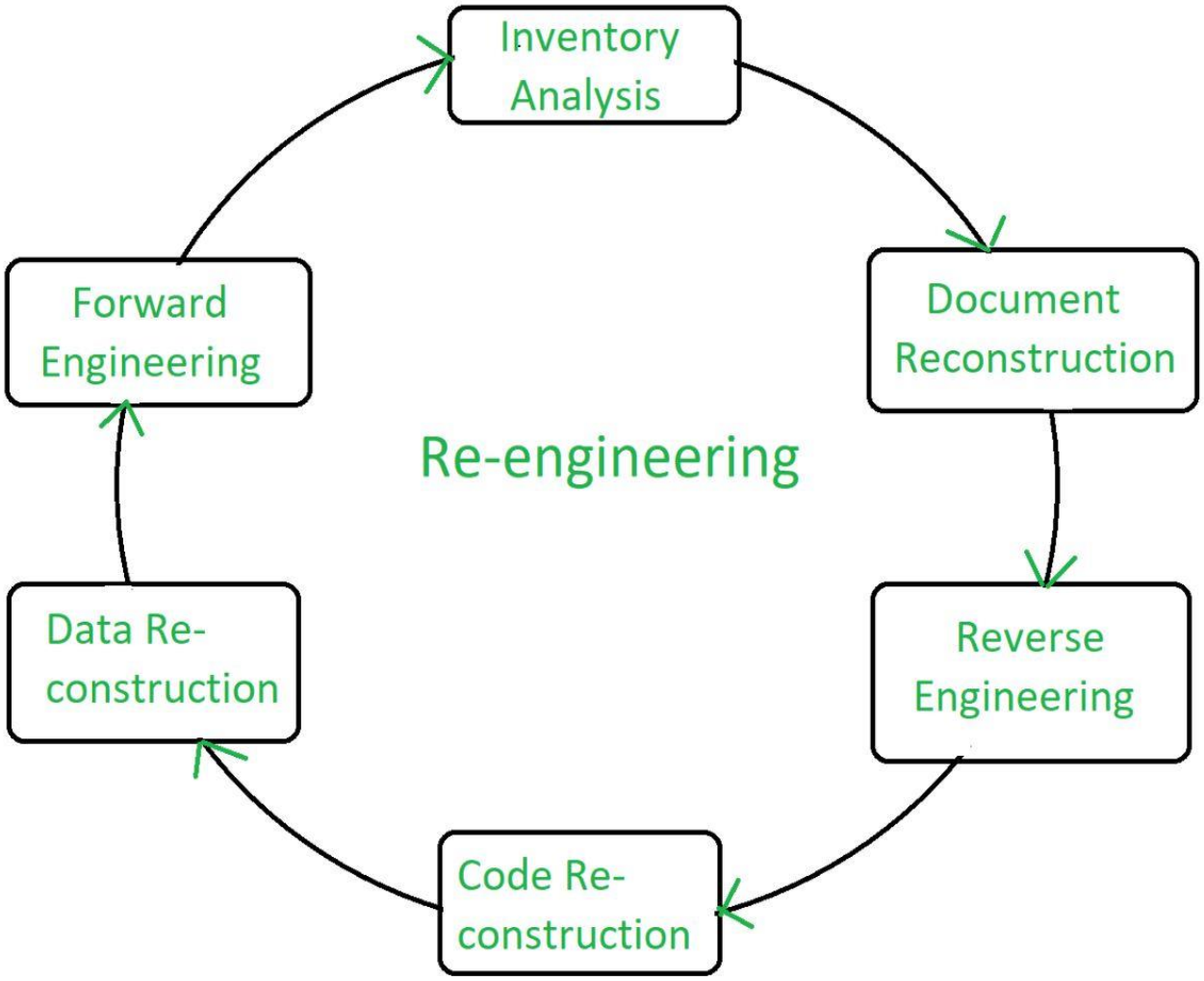
Re-engineering can be done for a variety of reasons, such as:

1. **To improve the software's performance and scalability:** By analyzing th existing code and identifying bottlenecks, re-engineering can be used to improve th software's performance and scalability.
2. **To add new features:** Re-engineering can be used to add new features o functionality to existing software.
3. **To support new platforms**: Re-engineering can be used to update existin software to work with new hardware or software platforms.
4. **To improve maintainability:** Re-engineering can be used to improve th software's overall design and architecture, making it easier to maintain and updat over time.
5. **To meet new regulations and compliance**: Re-engineering can be done to ensur that the software is compliant with new regulations and standards.
6. **Improving software quality:** Re-engineering can help improve the quality o software by eliminating defects, improving performance, and enhancing reliabilit and maintainability.
7. **Updating technology:** Re-engineering can help modernize the software system b updating the technology used to develop, test, and deploy the system.
8. **Enhancing functionality:** Re-engineering can help enhance the functionality of th software system by adding new features or improving existing ones.
9. **Resolving issues:** Re-engineering can help resolve issues related to scalabilit

security, or compatibility with other systems.

Steps involved in Re-engineering

1. **Inventory Analysis**
2. **Document Reconstruction**
3. **Reverse Engineering**
4. **Code Reconstruction**
5. **Data Reconstruction**
6. **Forward Engineering**



Steps of Re-Engineering

Re-engineering Cost Factors

1. The quality of the software to be re-engineered.
2. The tool support available for re-engineering.
3. The extent of the required data conversion.
4. The availability of expert staff for re-engineering.

Factors Affecting Cost of Re-engineering

Re-engineering can be a costly process, and there are several factors that cai affect the cost of re-engineering a software system:

1. **Size and complexity of the software:** The larger and more complex the softwar system, the more time and resources will be required to analyze, design, and modif it.
2. **Number of features to be added or modified:** The more features that need to b added or modified, the more time and resources will be required.
3. **Tools and technologies used:** The cost of re-engineering can be affected by th tools and technologies used, such as the cost of software development tools and th cost of hardware and infrastructure.
4. **Availability of documentation:** If the documentation of the existing system is no available or is not accurate, then it will take more time and resources to understan the system.
5. **Team size and skill level:** The size and skill level of the development team can als affect the cost of re-engineering. A larger and more experienced team may be able t complete the project faster and with fewer resources.
6. **Location and rate of the team:** The location and rate of the development team ca also affect the cost of re-engineering. Hiring a team in a lower-cost location or wit lower rates can help to reduce the cost of re-engineering.
7. **Testing and quality assurance:** Testing and quality assurance are importan aspects of re-engineering, and they can add significant costs to the project.
8. **Post-deployment maintenance:** The cost of post-deployment maintenance such a bug fixing, security updates, and feature additions can also play a role in the cost o re-engineering.

In summary, the cost of re-engineering a software system can vary depending on variety of factors, including the size and complexity of the software, the number o features to be added or modified, the tools and technologies used, and th availability of documentation and the skill level of the development team. It' important to carefully consider these factors when estimating the cost of re engineering a software system.

Advantages of Re-engineering

1. **Reduced Risk:** As the software is already existing, the risk is less as compared t new software development. Development problems, staffing problems an specification problems are the lots of problems that may arise in new softwar development.

2. **Reduced Cost:** The cost of re-engineering is less than the costs of developing nev software.
3. **Revelation of Business Rules:** As a system is re-engineered , business rules tha are embedded in the system are rediscovered.
4. **Better use of Existing Staff:** Existing staff expertise can be maintained an extended accommodate new skills during re-engineering.
5. **Improved efficiency:** By analyzing and redesigning processes, re-engineering ca lead to significant improvements in productivity, speed, and cost-effectiveness.
6. **Increased flexibility:** Re-engineering can make systems more adaptable t changing business needs and market conditions.
7. **Better customer service:** By redesigning processes to focus on customer needs, re engineering can lead to improved customer satisfaction and loyalty.
8. **Increased competitiveness:** Re-engineering can help organizations become mor competitive by improving efficiency, flexibility, and customer service.
9. **Improved quality:** Re-engineering can lead to better quality products and service by identifying and eliminating defects and inefficiencies in processes.
10. **Increased innovation:** Re-engineering can lead to new and innovative way of doing things, helping organizations to stay ahead of their competitors.
11. **Improved compliance:** Re-engineering can help organizations to compl with industry standards and regulations by identifying and addressing areas of non compliance.

Disadvantages of Re-engineering

Major architectural changes or radical reorganizing of the systems dat management has to be done manually. Re-engineered system is not likely to be a maintainable as a new system developed using modern software Re-engineerin methods.

1. **High costs:** Re-engineering can be a costly process, requiring significan investments in time, resources, and technology.
2. **Disruption to business operations:** Re-engineering can disrupt normal busines operations and cause inconvenience to customers, employees and othe stakeholders.
3. **Resistance to change:** Re-engineering can encounter resistance from employee who may be resistant to change and uncomfortable with new processes an technologies.
4. **Risk of failure:** Re-engineering projects can fail if they are not planned an executed properly, resulting in wasted resources and lost opportunities.
5. **Lack of employee involvement:** Re-engineering projects that are not properl communicated and involve employees, may lead to lack of employee engagemen and ownership resulting in failure of the project.
6. **Difficulty in measuring success:** Re-engineering can be difficult to measure i terms of success, making it difficult to justify the cost and effort involved.
7. **Difficulty in maintaining continuity:** Re-engineering can lead to significan changes in processes and systems, making it difficult to maintain continuity an

consistency in the organization.

SOFTWARE REUSE

oftware products are expensive. Therefore, software project managers are always vorried about the high cost of software development and are desperately looking for vays to cut development cost. A possible way to reduce development cost is to reuse arts from previously developed software. In addition to reduced development cost and ime, reuse also leads to higher quality of the developed products since the reusable omponents are ensured to have high quality. A reuse approach that is of late gaining rominence is component-based development. Component-based software development s different from the traditional software development in the sense that software is leveloped 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.

4.1 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

4.2 WHY ALMOST NO REUSE SO FAR?

A common scenario in many software development industries is explained further. Engineers working in software development organisations often have a feeling that the current system that they are developing is similar to the last few systems built. However, no attention is paid on how not to duplicate what can be reused from previously developed systems. Everything is being built from scratch. The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.

4.3 BASIC ISSUES IN ANY REUSE PROGRAM

The following are some of the basic issues that must be clearly understood for starting any reuse program:

- Component creation.
- Component indexing and storing. •

Component search.

- Component understanding.
- Component adaptation.
- Repository maintenance.

Component creation: For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. In Section 14.4, we discuss domain analysis as a promising technique which can be used to create reusable components.

Component indexing and storing

ndexing requires classification of the reusable components so that they can be easily earched when we look for a component for reuse. The components need to be stored in *relational database management system* (RDBMS) or an *object-oriented database ystem* (ODBMS) for efficient access when the number of components becomes large.

:omponent searching

'he programmers need to search for right components matching their requirements in a latabase of components. To be able to search components efficiently, the programmers equire a proper method to describe the components that they are looking for.

:omponent understanding

The programmers need precise and sufficiently complete understanding of what the omponent does to be able to decide whether they can reuse the component. To facilitate nderstanding, the components should be well documented and should do something imple.

:omponent adaptation

)ften, the components may need adaptation before they can be reused, since a selected omponent may not exactly fit the problem at hand. However, tinkering with the code is lso not a satisfactory solution because this is very likely to be a source of bugs.

:epository maintenance

A component repository once is created requires continuous maintenance. New omponents, as and when created have to be entered into the repository. The faulty omponents have to be tracked. Further, when new applications emerge, the older pplications become obsolete. In this case, the obsolete components might have to be emoved from the repository.

EMERGING TRENDS

We had discussed in Chapter 1 that software engineering techniques have in the past volved in response to the challenges posed to program development by the changing nvironment in which the programs run and also the changes to the types of applications equired by the users. By changes to the environment, we mean the changes that occur o the different technologies that underlie computer hardware, system software, etworking, and peripheral devices. Let us examine the way the environment has hanged of late. This can indicate the challenges being posed to the software levelopment principles. This in turn would give us some insight into the way in which he software engineering techniques are evolving of late.

The important changes to the environment that have occurred in the last two

decades include the following:

- The prices of computers have dropped drastically in this period. At the same time, they have become more powerful. Now they can perform computations much faster and store much larger volumes of data. The sizes of computers have shrunk and laptops and palmtops are becoming popular.
- The Internet has become extremely popular. Internet connects millions of computers world-wide and makes enormous available to the users.
- Networking techniques have made rapid progress. The speed of data transfer has increased unbelievably and at the same time, the cost of networking computers has dropped dramatically. Just to give an example of currently supported speed of data transfer, desktops now come with a default 1Gbps network port.
- Mobile phones have dramatically captured imagination of all. The level of acceptance that mobile phones have achieved in less than a decadeappears like a chapter straight out of a science fiction book. Mobile phones are rapidly transforming themselves into handheld computing devices. In addition to high speed fixed line connections, GPRS and wireless LANs have become common place.
- Over the last decade, cloud computing has become popular. In cloud computing, applications are hosted on cloud operating on a data center. Cloud computing is becoming more and more popular as it helps a user run sophisticated applications without much upfront investments and also frees him from buying and maintaining sophisticated hardware and software.

Challenges faced by software developers

Following are some of the challenges that are being faced by software developers:

- To cope up with fierce competitions, business houses are rapidly changing their business processes. This requires rapid changes to also occur to the software that support the business process activities. Therefore, there is a pressing demand to shorten the software delivery time. However, software is still taking unacceptably long time to develop and is turning out to be a bottleneck in implementing rapid business process changes. To reduce the software delivery times, software is being developed by teams working from globally distributed locations. How software can be effectively developed using globally distributed development teams is not yet clear and poses many challenges. On the other hand, radical changes to the software development principles are being put forward to shorten the development time.
- Business houses are getting tired of astronomical software costs, late deliveries, and poor quality products. On the other hand, hardware costs are dropping and

at the same time hardware is becoming more powerful, sophisticated, and reliable. Hardware and software cost differentials are becoming more and more glaring. The wisdom of developing every software from scratch is being questioned.Also,

alternate software delivery models are being proposed to reduce the software cost.

- Software sizes are further increasing.
- After Internet has become vastly popular, many software products are now required to interface with the Internet. Many products are even expected to work across the Internet. Also, with the availability of fast networks, distributed applications are becoming common place. However,it is not clear that how software is to be effectively developed in the context of distributed platforms and Internet.In response to the challenges faced, the following software engineering trends are becoming noticeable:

- Client-server software
- Service-oriented architecture (SOA) •

Software as a service (SaaS)

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 importan reasons are as follows:

Concurrency: A client-server software divides the computing work amongman different client and server components that could be residing on different machines Thus client-server solutions are inherently concurrent and as a result offer th advantage of faster processing.

Loose coupling: Client and server components are inherently loosely- coupled, 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 client- server 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-tierarchitectur hown in Figure 15.1(a). In a two-tier architecture, any client can get service from an erver 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.

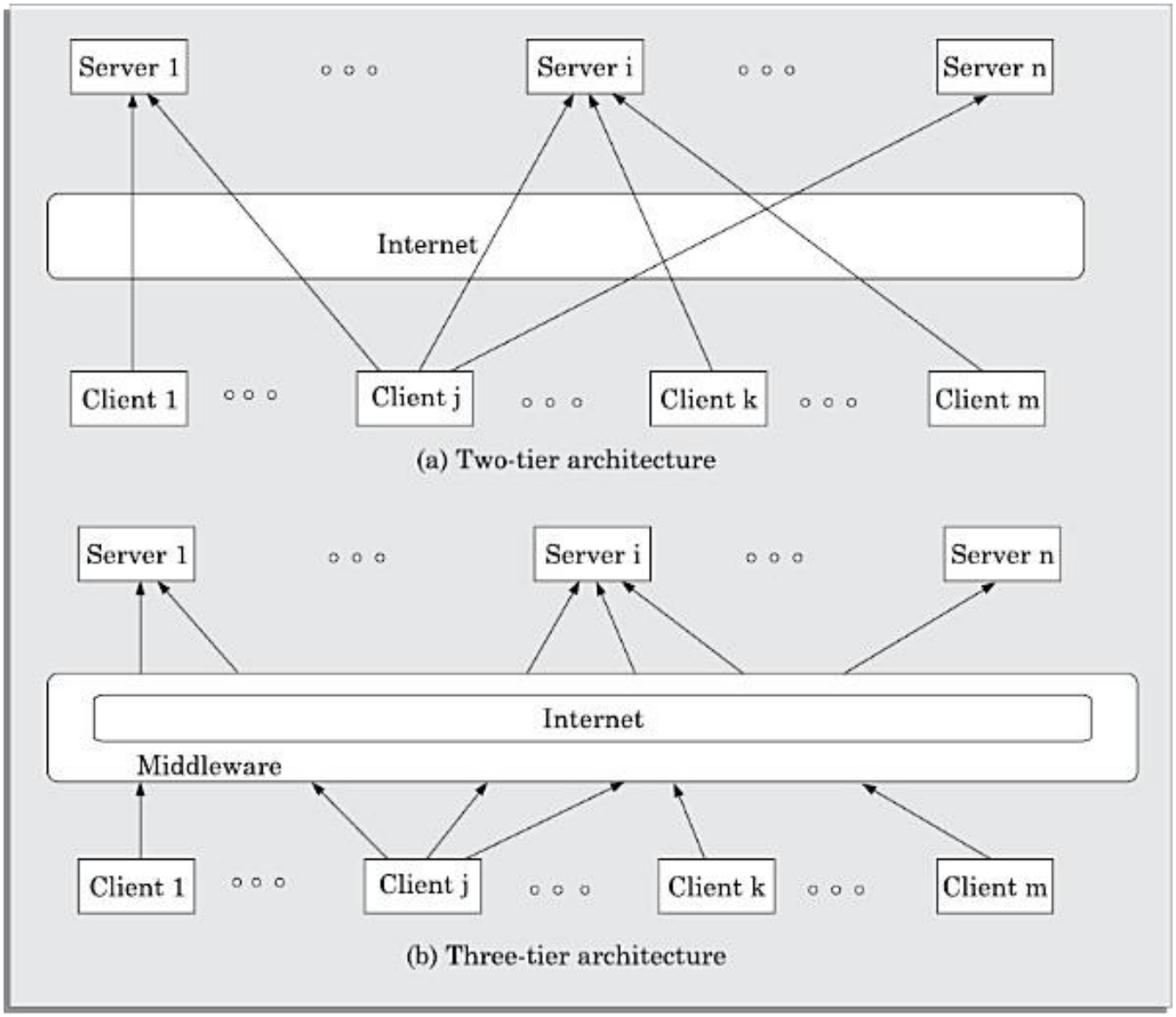


Figure 15.1: Two-tier and three-tier client-server architectures.

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 on- line 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 standardisations 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.

SOFTWARE AS A SERVICE (SAAS)

)wning software is very expensive. For example, a Rs. 50 Lakh software running on an ₹s. 1 Lakh computer is common place. As with hardware, owning software is the current radition across individuals and business houses. Most of IT budget now goes in upporting 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 er week. In this situation, it would be economically worthwhile to pay per hour of sage. This would also free the user from the botherance of maintenance, up gradation, ackup, etc. This is exactly what is advocated by SaaS.As we can see, SaaS shifts ownership" of the software from the customer to a service provider. Software owner rovides maintenance, daily technical operation, and support for the software. Services re 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 lso shifts the responsibility for hardware and software management from the customer o the provider. The cost of providing software services reduces as more and more ustomers 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.

- SaaS is changing the way software is delivered.
- SOA would fundamentally change the way we construct software systems. In the SOA paradigm, an application can be built by orchestrating existing services, and writing only the missing ones.