CENTRE FOR DISTANCE AND ONLINE EDUCATION

ADVANCED SOFTWARE ENGINEERING

M.Sc. Computer Science

MANONMANIAM SUNDARANAR UNIVESITY
TIRUNELVELI

Course code		ADVANCED SOFTWARE ENGINEERING	L	T	P	C
Core/Elective/Supportive		Elective	3			3
Pre-requisite		Basics of Software Engineering & SPM				

Course Objectives:

The main objectives of this course are to:

1. Introduce to Software Engineering, Design, Testing and Maintenance.
2. Enable the students to learn the concepts of Software Engineering.
3. Learn about Software Project Management, Software Design & Testing.

Expected Course Outcomes:

On the successful completion of the course ,student will be able to:

1	Understand about Software Engineering process	K1,K2
2	Understand about Software project management skills, design and quality management	K2,K3
3	Analyze on Software Requirements and Specification	K3,K4
4	Analyze on Software Testing, Maintenance and Software Re-Engineering	K4,K5
5	Design and conduct various types and levels of software quality for a software project	K5,K6

K1-Remember;**K2**-Understand;**K3**-Apply;**K4**-Analyze;**K5**-Evaluate; **K6**-Create

Unit:1	INTRODUCTION	15hours

Introduction: The Problem Domain – Software Engineering Challenges - Software Engineering Approach – Software Processes: Software Process – Characteristics of a Software Process – Software Development Process Models – Other software processes.

Unit:2	SOFTWARE REQUIREMENTS	15hours

Software Requirements Analysis and Specification : Requirement engineering – Type of Requirements – Feasibility Studies – Requirements Elicitation – Requirement Analysis – Requirement Documentation – Requirement Validation – Requirement Management – SRS - Formal System Specification – Axiomatic Specification – Algebraic Specification - Case study: Student Result management system. Software Quality Management –Software Quality, Software Quality Management System, ISO 9000, SEI CMM.

Unit:3	PROJECT MANAGEMENT	15hours
Software Project Management: Responsibilities of a software project manager – Project planning – Metrics for Project size estimation – Project Estimation Techniques – Empirical Estimation Techniques – COCOMO – Halstead's software science – Staffing level estimation – Scheduling– Organization and Team Structures – Staffing – Risk management – Software Configuration Management – Miscellaneous Plan.		
Unit:4	**SOFTWARE DESIGN**	**15hours**
Software Design: Outcome of a Design process – Characteristics of a good software design – Cohesion and coupling - Strategy of Design – Function Oriented Design – Object Oriented Design - Detailed Design - IEEE Recommended Practice for Software Design Descriptions.		
Unit:5	**SOFTWARE TESTING**	**13hours**
Software Testing: A Strategic approach to software testing – Terminologies – Functional testing– Structural testing – Levels of testing – Validation testing - Regression testing – Art of Debugging–Testingtools-Metrics-ReliabilityEstimation.SoftwareMaintenance -Maintenance Process - Reverse Engineering – Software Re-engineering - Configuration Management Activities.		
Unit:6	**Contemporary Issues**	**2 hours**
Expert lectures, online seminars –webinars		

Unit I

INTRODUCTION

What is software engineering?

- "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 two these 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.

Is software engineering a science or an art?

- Just as any other engineering discipline, software engineering makes heavy use of the knowledge that has accrued from the experiences of a larges number of practitioners. These past experiences have been systematically organized and wherever possible theoretical basis to the empirical observations have been provided. Whenever no reasonable theoretical justification could be provided, the past experiences have been adopted as rule of thumb. In contrast, all scientific solutions are constructed through rigorous application of provable principles.
- As is usual in all engineering disciplines, in software engineering several conflicting goals are encountered while solving a problem. In such situations, several alternate solutions are first proposed. An appropriate solution is chosen out of the candidate solutions based on various trade-offs that need to be made on account of issues of cost, maintainability, and usability. Therefore, while arriving at the final solution, several iterations and are possible.
- Engineering disciplines such as software engineering make use of only well-understood and well-documented principles. Art, on the other hand, is often based on making subjective judgement based on qualitative attributes and using ill-understood principles.

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

1. EVOLUTION—FROM AN ART FORM TO AN ENGINEERING DISCIPLINE

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

(i) Evolution of an Art into an Engineering Discipline

- Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.
- The early programmers used an *ad hoc* programming style. This style of program development is now variously being referred to as *exploratory*, *build and fix*, and *code and fix* styles.
- **Build and Fix Style**
    - In a build and fix style, a program is quickly developed without making any specification, plan, or design. The different imperfections that are subsequently noticed are fixed.
- **Exploratory Programming Style**
    - The exploratory programming style is an informal style in the sense that there are no set rules or recommendations.
    - Every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies.
    - The exploratory style comes naturally to all first-time programmers.
    - The exploratory style usually yields poor quality and un maintainable code and also makes program development very expensive as well as time-consuming.
- As we have already pointed out, the build and fix style were widely adoptedby the programmers in the early years of computing history.
- We can consider the exploratory program development style as an art—

since this style, as is the case with any art, is mostly guided by intuition.

- There are many stories about programmers in the past who were like proficient artists and could write good programs using an essentially build and fix model and some esoteric knowledge.
- In contrast, the programmers working in modern software industry rarely make use of any esoteric knowledge and develop software by applying some well-understood principles.

2. SOFTWARE DEVELOPMENT PROJECTS

(i) Programs *versus* Products

Program	Product
Set of instruction related each other	Collection of programs designed for specific task.
software is being developed by individuals such as students for their classroom assignments and hobbyists for their personal use.	A professional software is developed by a group of software developers working together in a team.
These are usually small in size	professional software is often too large and complex to be developed by any single individual.
limited functionalities	Many numbers of functionalities
The author of a program is usually the sole user of the software	A software product has a large number of users, it is systematically designed, carefully implemented, and thoroughly tested.
Lack of good user-interface	Good User Interface
Lack of proper Documentation	Good Documentation
Poor maintainability, efficiency, and reliability.	Reliable, Efficiency and Maintainability
Do not have any supporting documents such as users' manual, maintenance manual, design document, test documents, etc.	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.

(ii) Types of Software Development Projects

-Generic products:

- [This type of software product are developed by a organization and sold on open market to any customer], (System software, application software)
- We all know of a variety of software such as Microsoft's Windows and the

Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc.

- This software is available off-the-shelf for purchase and are used by a diverse range of customers. These are called *generic software products* since many users essentially use the same software.
- These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own.

-Customized (or bespoke) products:

- This type of software products is developed by a software contractor and especially for a customer.

Software services

- A software service usually involves either development of a *customized software* or development of some specific part of a software in an outsourced mode.
- A *customized software* is developed according to the specification drawn up by one or at most a few customers. These need to be developed in a short time frame (typically a couple of months), and at the same time the development cost must be low. Usually, a developing company develops customized software by tailoring some of its existing software.
- Another type of software service is *outsourced software*. Sometimes, it can make good commercial sense for a company developing a large project to outsource some parts of its development work to other companies.

-**Embedded Product:** Combination of both hardware and software

3. EMERGENCE OF SOFTWARE ENGINEERING

(i) Early Computer Programming

- Early commercial computers were very slow and too elementary as compared to today's standards. Even simple processing tasks took considerable computation time on those computers.

- Those programs were usually written in assembly languages.
- Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code.
- Every programmer developed his own individualistic style of writing programs according to his intuition and used this style ***ad hoc*** while writing different programs.
- In simple words, programmers wrote programs without formulating any proper solution strategy, plan, or design a jump to the terminal and start coding immediately on hearing out the problem.
- They then went on fixing any problems that they observed until they had a program that worked reasonably well. We have already designated this style of programming as **the *build and fix* (or the *exploratory programming)* style.**

(ii) High-level Language Programming

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

(iii) Control Flow-based Design

- As the size and complexity of programs kept on increasing, experienced programmers advised other programmers to pay particular attention to the design of a **program's *control flow structure*.**
- A program's control flow structure indicates the sequence in which the program's instructions are executed.
- In order to help develop programs having good control flow structures, the ***flow-charting*** *technique* was developed.
- E v e n today, t h e flow charting technique is being used to represent and design algorithms; though the popularity of flow charting represent and design programs has want to a great extent due to the emergence of more advanced techniques.
- Figure 1.8 illustrates two alternate ways of writing program code for the same problem. The flow chart representations for the two program segments of Figure 1.8 are drawn in Figure 1.9.
- Observe that the control flow structure of the program segment in Figure 1.9(b) is much simpler than that of Figure 1.9(a). By examining thecode, it can be seen that Figure 1.9(a) is much harder to understand as compared to Figure 1.9(b).
- This example corroborates the fact that if the flow chart representation is simple, then the corresponding code should be simple.
- You can draw t h e flow chart representations of several other problems to convince yourself that a program with complex flow chart representation is indeed more difficult to understand and maintain.

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

(a) An example unstructured program

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

(b) Corresponding structured program

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

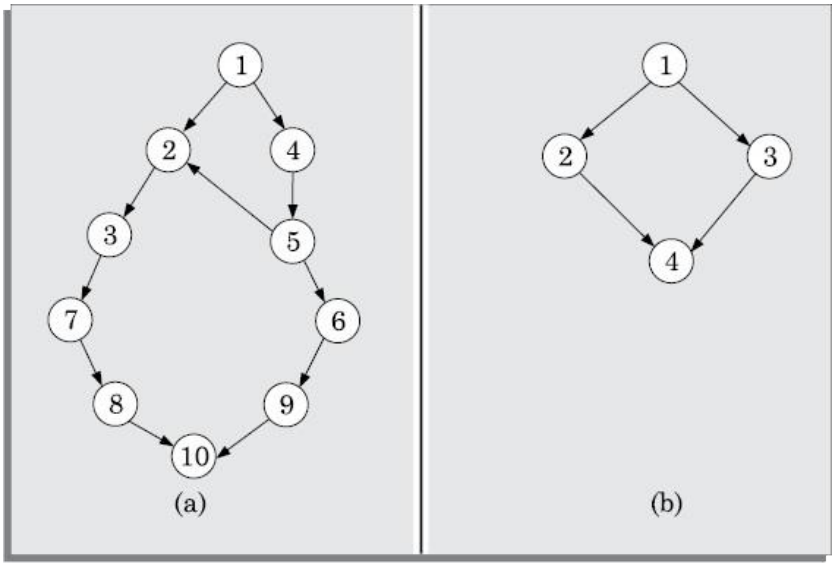


Figure 1.9: Control flow graphs of the programs of Figures 1.8(a) and (b).

- Let us now try to understand why a program having good control flow structure would be easier to develop and understand.
- we may start with the input data and check by running through the program how each statement processes (transforms) the input data until the output is produced.
- For example, for the program of Fig 1.9(a) you would have to understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and 1- 4-5-2-3-7-8-10.
- A program having a messy control flow (i.e. flow chart) structure, would have a large number of execution paths (see Figure 1.10). Consequently, it would become extremely difficult to determine all the execution paths, and tracing the execution sequence along all the paths trying to understand them can be nightmarish. It is therefore evident that a program having a messy flow chart representation would indeed be difficult to understand and debug.

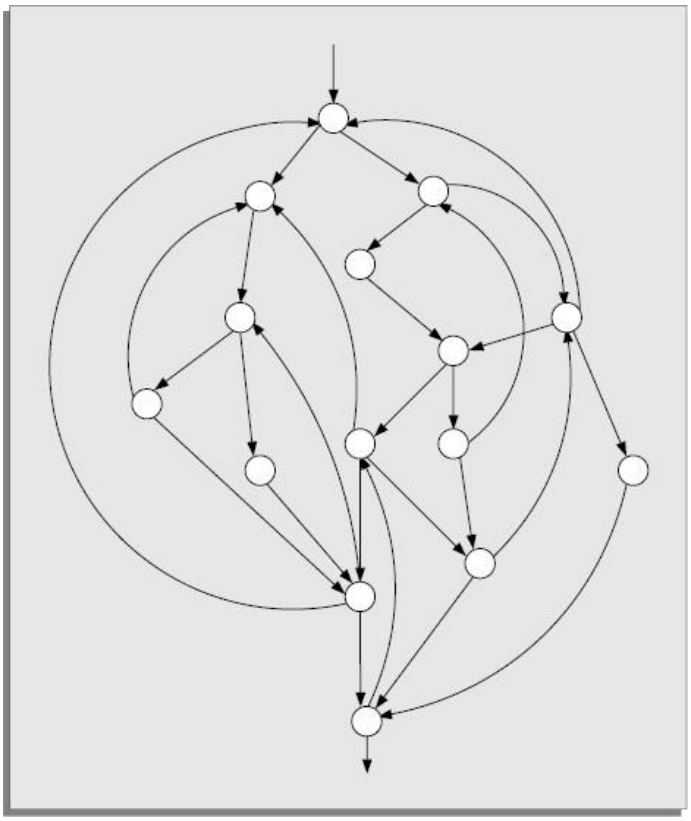


Figure 1.10: CFG of a program having too many GO TO statements.

Are GO TO statements the culprits?

- GO TO statements alter the flow of control arbitrarily, resulting in too many paths. But, then why does use of too many GO TO statements makes a program hard to understand?
- A programmer trying to understand a program would have to mentally trace and understand the processing that take place along all the paths of the program making program understanding and debugging extremely complicated.
- Soon it became widely accepted that good programs should have very simple control structures
- The use of flow charts to design good control flow structures of programs became wide spread.

Structured programming—a logical extension

- The need to restrict the use of GO TO statements was recognised by everybody.
- However, many programmers were still using assembly languages. JUMP instructions are frequently used for program branching in assembly languages.
- Bohm and Jacopini that only three programming constructs—sequence, selection, and iteration—were sufficient to express any programming logic.
- An example of a **sequence statement** is an assignment statement of the form a=b;.
- Examples **of selection and iteration statements are** the **if-then-else and the do-while statements** respectively.

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

- Structured programs avoid unstructured control flows by restricting the use of GO TO statements.

- Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, an important feature of structured programs is the design of good control structures.

- An example illustrating this key difference between structured and unstructured programs is shown in Figure 1.8. The program in Figure 1.8(a) makes use of too many GO TO statements, whereas the program in Figure 1.8(b) makes use of none. The flow chart of the program making use of GO TO statements is obviously much more complex as can be seen in Figure 1.9.
- Besides the control structure aspects, the term *structured program* is being used to denote a couple of other program features as well.
- A structured program should be **modular**. A modular program is one

which is decomposed into a set of modules such that the modules should have low interdependency among each other.

- programmers commit a smaller number of errors while using structured if- then-else and do-while statements than when using test-and-branch code constructs.
- Besides being less error-prone, structured programs are normally more readable, easier to maintain, and require less effort to develop compared to unstructured programs.
- Very soon several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support structured programming.

(iv) Data Structure-oriented Design

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

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

- In the next step, the program design is derived from the data structure.
- An example **of a data structure-oriented design technique is the Jackson's Structured Programming (JSP)** technique developed by Michael Jackson [1975].

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

(v) Data Flow-oriented Design

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

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

- The functions (also called **as *processes***) and the data items that are exchanged between the different functions are represented in a diagram known as a *data flow diagram* (DFD).

- The program structure can be designed from the DFD representation of the problem.
- **DFDs: A crucial program representation for procedural program design**
- **DFD has proven to be a generic technique which is being used to**

model all types of systems, and not just software systems.

- **Each circle in the DFD** model represents a ***process*** **or** ***bubble***.
- Each process consumes certain input items and produces certain output items.
- Once you develop the DFD model of a problem, data flow-oriented design techniques provide a rather straight forward methodology to transform the DFD representation of a problem into an appropriate software design.

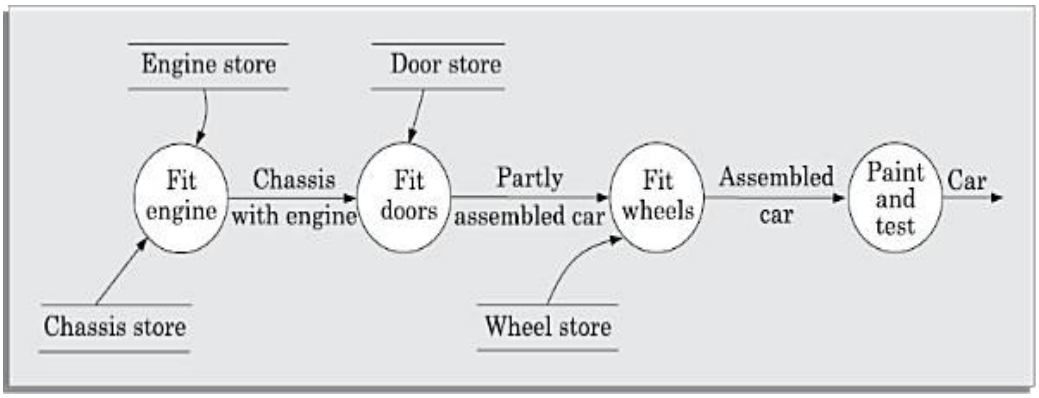


Figure 1.11: Data flow model of a car assembly plant.

(vi) Object-oriented Design

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

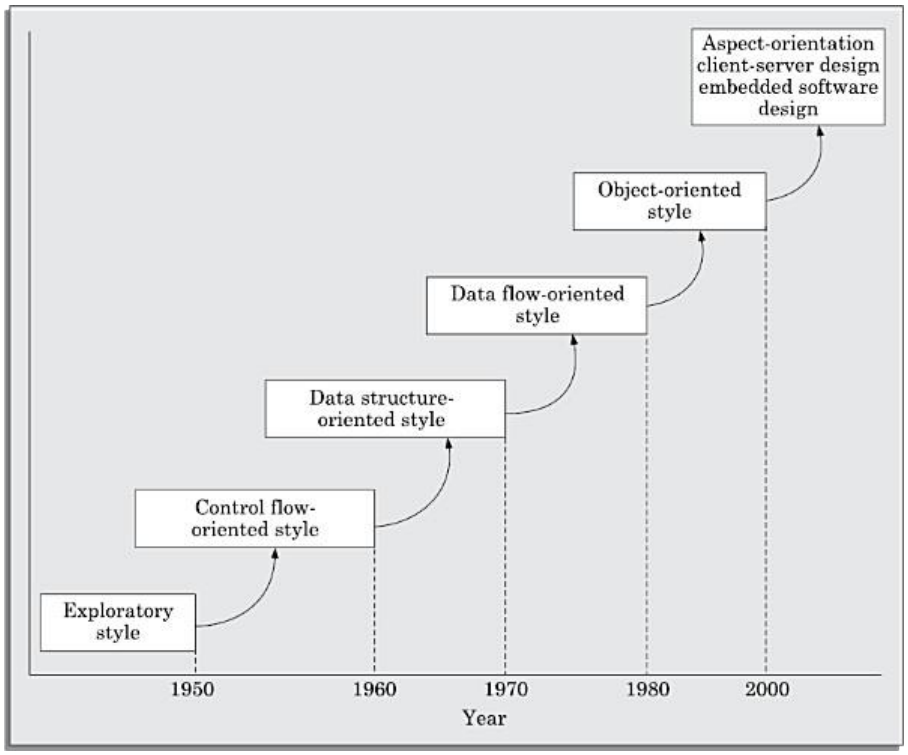


Figure 1.12: Evolution of software design techniques.

4. SOFTWARE LIFE CYCLE

- The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.
- **This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the *inception* stage**.
- Starting with the inception stage, a software evolves through a series of identifiable stages (also called **phases**) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.
- Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called ***maintenance***) phase.
    - As the users use the software, not only do they request for fixing any failures that they might encounter, but they also continually suggest several improvements and modifications to the software.

Thus, the maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user. The operation phase is usually the longest of all phases and constitutes the useful life of a software.

- Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of a new software having improved features and working, changed computing platforms, etc.

Software development life cycle (SDLC) model

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

Process *versus* methodology

- The term **process** has a broader scope and addresses either all **the activities taking place during software development, or certain coarse grained activities such as design (e.g. design process), testing (test process)**, etc.
- Further, a software process not only identifies the specific activities that need to be carried out, but may also prescribe certain methodology for carrying out each activity.
- A **methodology**, on the other hand, prescribes a set of steps for carrying out a specific life cycle activity.
- It may also include the rationale and philosophical assumptions behind the set of steps through which the activity is accomplished.

Why use a development process?

- A software development process has a much broader scope as compared to a software development methodology
- A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a

chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity.

- A methodology, in contrast, describes the steps to carry out only a singleor at best a few individual activities.
- The primary advantage of using a development process is that it encourages development of software in a systematic and disciplined manner
- Software development organisations have realised that adherence to a suitable life cycle model helps to produce good quality software and that helps minimise the chances of time and cost overruns.
- Adhering to a process is especially important to the development of professional software needing team effort.
- When software is developed by a team rather than by an individual programmer, use of a life cycle model becomes indispensable for successful completion of the project.
- The difficulties that may arise if a team does not use any development process, and the team members are given complete freedom to develop their assigned part of the software as per their own discretion.
- Therefore, *ad hoc* development turns out to be is a sure way to have a failed project. Believe it or not, this is exactly what has caused many project failures in the past!
- When a software is developed by a team, it is necessary to have a precise understanding among the team members as to—when to do what. In the absence of such an understanding, if each member at any time would do whatever activity he feels like doing. This would be an open invitation to developmental chaos and project failure.
- The use of a suitable life cycle model is crucial to the successful completion of a team-based development project. But, do we need an SDLC model for developing a small program. In this context, we need to distinguish between programming-in-the-small and programming-in-the-large.
- **Programming-in-the-small refers to development of a toy program by a single programmer. Whereas programming-in-the-large**

refers to development of a professional software through team effort. While development of a software of the former type could succeed even while an individual programmer uses a build and fix style of development, use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.

Why document a development process?

- It is not enough for an organisation to just have a well-defined development process, but t h e development process needs to be properly documented.
- In this case, its developers develop o n l y an informal understanding of the development process.
- An informal understanding of the development process among the team members can create several problems during development.
- A documented process model ensures that every activity in the life cycle is accurately defined.
- Also, wherever necessary the methodologies for carrying out the respective activities are described.
- Without documentation, the activities and their ordering tend to be loosely defined, leading to confusion and misinterpretation by different teams in the organisation.
- For example, code reviews may informally and inadequately be carried out since there is no documented methodology as to how the code review should be done. Another difficulty is that for loosely defined activities, the developers tend to use their subjective judgments. As an example, unless it is explicitly prescribed, the team members would subjectively decide as to whether the test cases should be designed just after the requirements phase, after the design phase, or after the coding phase. Also, they would debate whether the test cases should be documented at all and the rigour with it should be documented.
- An undocumented process gives a clear indication to the members of the development teams about the lack of seriousness on the part of the management of the organisation about following the process.

- Therefore, an undocumented process serves as a hint to the developers to loosely follow the process. The symptoms of an undocumented process are easily visible—designs are shabbily done, reviews are not carried out rigorously, etc.
- A project team might often have to tailor a standard process model for use in a specific project. It is easier to tailor a documented process model, when it is required to modify certain activities or phases of the life cycle.
- A documented process model would help to identify where exactly the required tailoring should occur.
- A documented process model, is a mandatory requirement of the modern quality assurance standards such as ISO 9000 and SEI CMM. This means that unless a software organisation has a documented process, it would not qualify for accreditation with any of the quality standards.
- In the absence of a quality certification for the organisation, the customers would be suspicious of its capability of developing quality software and the organisation might find it difficult to win tenders for software development.

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

- Nowadays, good software development organisations normally document their development process in the form of a booklet.

- T h e y expect the developers recruited fresh to their organisation to first master their software development process during a short induction training that they are made to undergo.

Phase entry and exit criteria

- A good SDLC besides clearly identifying the different phases in the life

cycle, should unambiguously **define the entry and exit criteria for each phase.**

- The phase entry (or exit) criteria is usually expressed as a set of conditions that needs to be be satisfied for the phase to start (or to complete).
- As an example, the phase exit criteria for the software requirements specification phase, can be that the *software requirements specification* (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer. Only after these criteria are satisfied, the next phase can start.
- If the entry and exit criteria for various phases are not well-defined, then that would leave enough scope for ambiguity in starting and ending various phases, and cause lot of confusion among the developers.
- When the phase entry and exit criteria are not well-defined, the developers might close the activities of a phase much before they are actually complete, giving a false impression of rapid progress.
- In this case, it becomes very difficult for the project manager to determine the exact status of development and track the progress of the project. This usually leads to a problem that is usually identified as the *99 per cent complete syndrome*. This syndrome appears when there the software project manager has no definite way of assessing the progress of a project, the optimistic team members feel that their work is 99 per cent complete even when their work is far from completion—making all projections made by the project manager about the project completion time to be highly inaccurate.

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.

5. CLASSICAL WATERFALL MODEL

- Classical waterfall model is intuitively the most obvious way to develop software.
- It is simple but idealistic.
- All other life cycle models can be thought of as being extensions of the classical waterfall model.
- The classical waterfall model divides the life cycle into a set of phases as shown in Figure 2.1.

- It resembles a multi-level waterfall. This resemblance justifies the name of the model.

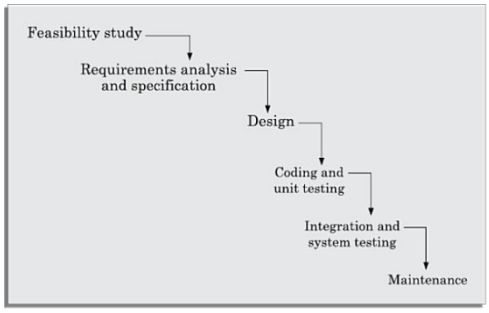


Figure 2.1: Classical waterfall model.

Phases of the classical waterfall model

- The different phases 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, changes to it become necessary on account of bug fixes and feature

extensions, causing maintenance works to be undertaken. Therefore, the last phase is also known as the ***maintenance phase*** of the life cycle.

- An activity that spans all phases of software development is ***project management***.
- **Project management**, is an important activity in the life cycle and deals with managing the software development and maintenance activities.
- In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team.
- The 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.

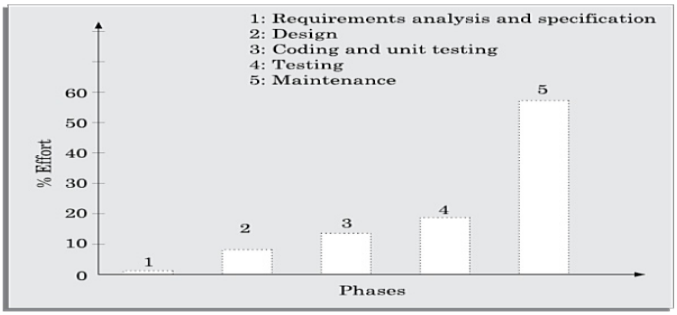


Figure 2.2: Relative effort distribution among different phases of a typical product.

Feasibility study

- The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software.
- The feasibility study involves carrying out several activities such as collection of basic information relating to the software such as the different data items that would be **input** to the system, the **processing** required to be carried out on these data, the **output** data required to be produced by the system, as well as various constraints on the development.
- These collected data are analysed to perform at the following:
  - **Development of an overall understanding of the problem:**
    - It is necessary to first develop an overall understanding of what the customer requires to be developed.
  - **Formulation of the various possible strategies for solving the**

problem:

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

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

- We can summarise the outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not, at this stage very high-level decisions regarding the solution strategy is defined. Therefore, feasibility study is a very crucial stage in software development.

Requirements analysis and specification

- The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly.
- This phase consists of two distinct activities, namely **requirements gathering and analysis, and requirements specification**.

  Requirements gathering and analysis:

  - 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 an ***inconsistent*** **requirement** is one in which some part of the requirement contradicts with some other part.
- On the other hand, an ***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.
- **The SRS document normally serves as a contract between the development team and the customer.**
- Any future dispute between the customer and the developers can be settled by examining the SRS document. The SRS document is therefore an important document which must be thoroughly understood by the development team, and reviewed jointly with the customer.
- The SRS document not only forms the basis for carrying out all the development activities, but several documents such as users' manuals, system test plan, etc. are prepared directly based on it.

Design

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

- **Procedural design approach:**
  - The traditional design approach is in use in many software development projects at the present time. This traditional design technique is based on the **data flow-oriented design a**pproach.
  - It consists of two important activities; first ***structured analysis*** of the requirements specification is carried out where the detailed structure of the problem is examined.
  - This is followed by a ***structured design*** step where the results of structured analysis are transformed into the software design.
  - During structured analysis, the **functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analysed and represented diagrammatically in the form of DFDs.**
  - Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities—**architectural design (also called *high-level design*) and detailed design (also called *Low-level design*).**
  - **High-level design involves** decomposing the system in to modules, and representing the interfaces and the invocation relationships among the modules. A high-level software design is some times referred to as the ***software architecture***.
  - **During the detailed design activity**, internals of the individual modules such as the data structures and algorithms of the modules are designed and documented.
- **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 the ***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.
- 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.

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

- System testing is carried out on this fully working system.
- System testing usually consists of three different kinds of testing

activities:

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

- β testing: This is the system testing performed by a friendly set of customers.

-Acceptance testing: After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

Maintenance

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

  Corrective maintenance: This type of maintenance is carried out to correct errors that were not discovered during the product development phase.

  Perfective maintenance: This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.

  Adaptive maintenance: Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Shortcomings of the classical waterfall model

- Let us identify some of the important shortcomings of the classical waterfall model:
- **No feedback paths:**
  - 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.

- **Difficult to accommodate change requests:**
  - The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.
- **Inefficient error corrections:**
  - This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.
- **No overlapping of phases:**
  - This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes.
  - Consequently, it is safe to say that in a practical software development scenario, rather than having a precise point in time at which a phase transition occurs, the different phases need to overlap for cost and efficiency reasons.
- **Is the classical waterfall model useful at all?**
  - It is hard to use the classical waterfall model in real projects.
  - In any practical development environment, as the software takes shape, several iterations through the different waterfall stages become necessary for correction of errors committed during various phases. Therefore, the classical waterfall model is hardly usable for software development.

6. ITERATIVE WATERFALL MODEL

- The iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects.

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

- The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3.

- The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.

- For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to be reflected in the design documents and all other subsequent documents.

- 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

- 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.

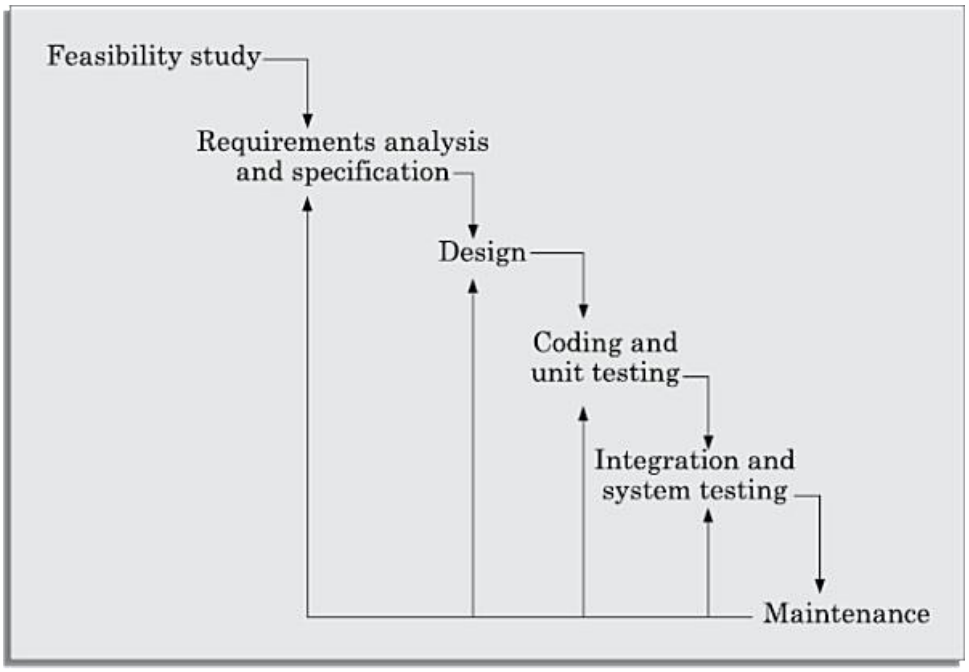


Figure 2.3: Iterative waterfa**l** model.

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* or *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 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 testing activities, thereby incurring higher cost. It may not always be possible to detect all the errors in the same phase in which they are made. Nevertheless, the errors should be detected as early as possible.

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

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

Phase overlap

- Even though the strict waterfall model envisages sharp transitions to occur from one phase to the next (see Figure 2.3), in practice the activities of different phases overlap (as shown in Figure 2.4) due to two main reasons:
    - In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. These subsequently detected errors cause the activities of some already completed phases to be reworked. If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure2.4.
    - An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members. Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a *blocking state*. Thus the developers who complete early would idle while waiting for their team mates to complete their assigned work. Clearly this is a cause for wastage of resources and a source of cost escalation and inefficiency. As a result, in real projects, the phases are allowed to overlap. That is, once a developer completes his work assignment for a phase, proceeds to start the work for the next phase, without waiting for all his team members to complete

their respective work allocations.

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

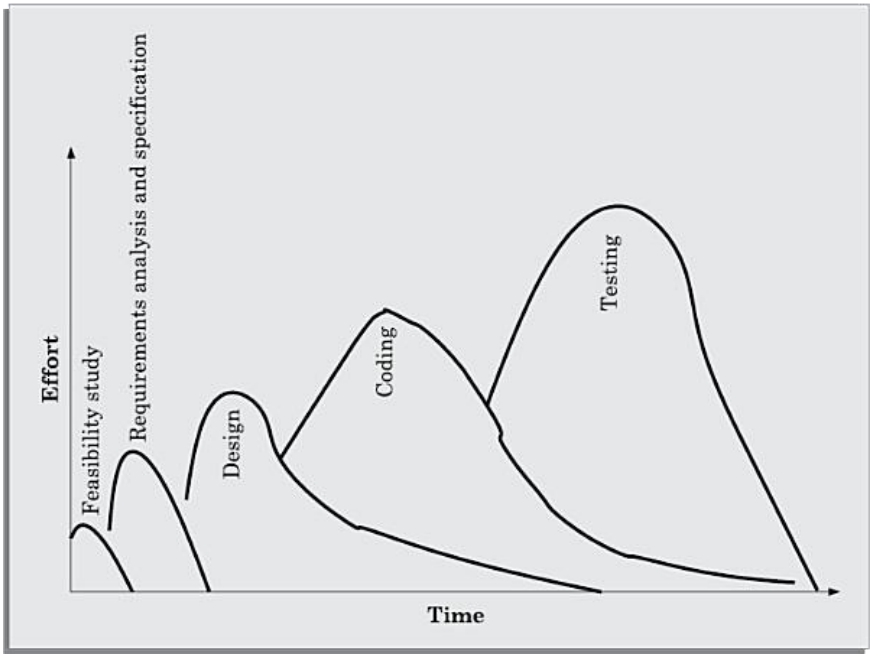


Figure 2.4: Distribution of effort for various phases in the iterative waterfa **l** model.

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. Now, not only software has become very large and complex, very few (if at all any) software project is being developed from scratch.
- **Difficult to accommodate change requests:** A major problem with the waterfall model is that the requirements need to be frozen before the development starts. Therefore, accommodating even small change requests after the development activities are underway not only requires overhauling the plan, but also the artifacts that have already been developed.

Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.

- While the waterfall model is inflexible to later changes to the requirements, evidence gathered from several projects points to the fact that later changes to requirements are almost inevitable. Even for projects with highly experienced professionals at all levels, as well as computer savvy customers, requirements are often missed as well as misinterpreted. Unless change requests are encouraged, the developed functionalities would be misfit to the true customer requirements.
- Requirement changes can arise due to a variety of reasons including the following—requirements were not clear to the customer, requirements were misunderstood, business process of the customer may have changed after the SRS document was signed off, etc. In fact, customers get clearer understanding of their requirements only after working on a fully developed and installed system.
- The basic assumption made in the iterative waterfall model that methodical requirements gathering and analysis alone would comprehensively and correctly identify all the requirements by the end of the requirements phase is flawed.
- **Incremental delivery not supported:** In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur.
- This is problematic because the complete application may take several months or years to be completed and delivered to the customer.
- By the time the software is delivered, installed, and becomes ready for use, the customer's business process might have changed substantially. This makes the developed application a poor fit to the customer's requirements.
- **Phase overlap not supported:** For most real life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model.
- By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*.

- The waterfall model is usually adapted for use in real-life projects by allowing overlapping of various phases as shown in Figure 2.4.
- **Error correction unduly expensive:** In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayeddelivery.
- **Limited customer interactions:** This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion. As a result, the developed software usually turns out to be a misfit to the customer's actual requirements.
- **Heavy weight:** The waterfall model overemphasises documentation. A significant portion of the time of the developers is spent in preparing documents, and revising them as changes occur over the life cycle. Heavy documentation though useful during maintenance and for carrying out review, is a source of team inefficiency.
- **No support for risk handling and code reuse:** It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of proj ects usually involve significant reuse.

7. V-MODEL

- A popular development process model, V-model is a variant of the waterfall model.
- As is the case with the waterfall model, this model gets its name from its visual appearance (see Figure 2.5).
- In this model verification and validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce.

- This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability.

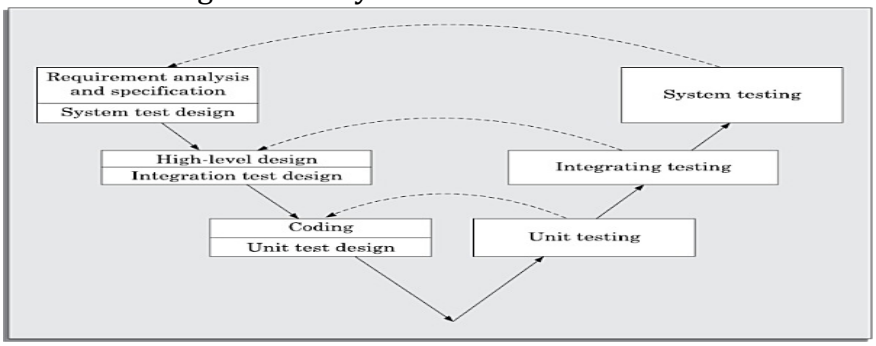


Figure 2.5: V-model.

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

    - In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which have been shown by dotted arcs in Figure 2.5.
    - In the validation phase, testing is carried out in three steps—unit, integration, and system testing. The purpose of these three different steps of testing during the validation phase is to detect defects that arise in the corresponding phases of software development— requirements analysis and specification, design, and coding respectively.

V-model *versus* waterfall model

- In contrast to the iterative waterfall model where testing activities are confined to the testing phase only, in the V-model testing activities are spread over the entire life cycle.
- As shown in Figure 2.5, during the requirements specification phase, the system test suite design activity takes place. During the design phase,

the integration test cases are designed. During coding, the unit test cases are designed. Thus, we can say that in this model, development and validation activities proceed hand in hand.

Advantages of V-model

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

- In the V-model, much of the testing activities (test case design, test planning, etc.) are carried out in parallel with the development activities. Therefore, before testing phase starts significant part of the testing activities, including test case design and test planning, is already complete. Therefore, this model usually leads to a shorter testing phase and an overall faster product development as compared to the iterative model.
- Since test cases are designed when the schedule pressure has not built up, the quality of the test cases are usually better.
- The test team is reasonably kept occupied throughout the development cycle in contrast to the waterfall model where the testers are active only during the testing phase. This leads to more efficient manpower utilisation.
- In the V-model, the test team is associated with the project from the beginning. Therefore they build up a good understanding of the development artifacts, and this in turn, helps them to carry out effective testing of the software. In contrast, in the waterfall model often the test team comes on board late in the development cycle,since no testing activities are carried out before the start of the implementation and testing phase.

Disadvantages of V-model

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

8. PROTOTYPING MODEL

- The prototype model is also a popular life cycle model.
- The prototyping model can be considered to be an extension of the waterfall model.
- This model suggests building a working *prototype* of the system, before development of the actual software.
- **A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software.**
- A prototype can be built very quickly by using several shortcuts.
- The shortcuts usually involve developing inefficient, inaccurate, or dummy functions.
- *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.

      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**.
- 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. As shown in Figure 2.6, software is developed through two major activities—**prototype construction and iterative waterfall-based software development.**
- **Prototype development:** Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.
- **Iterative development:** Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the working prototype

that has been approved by the customer serves as an animated requirements specification.

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

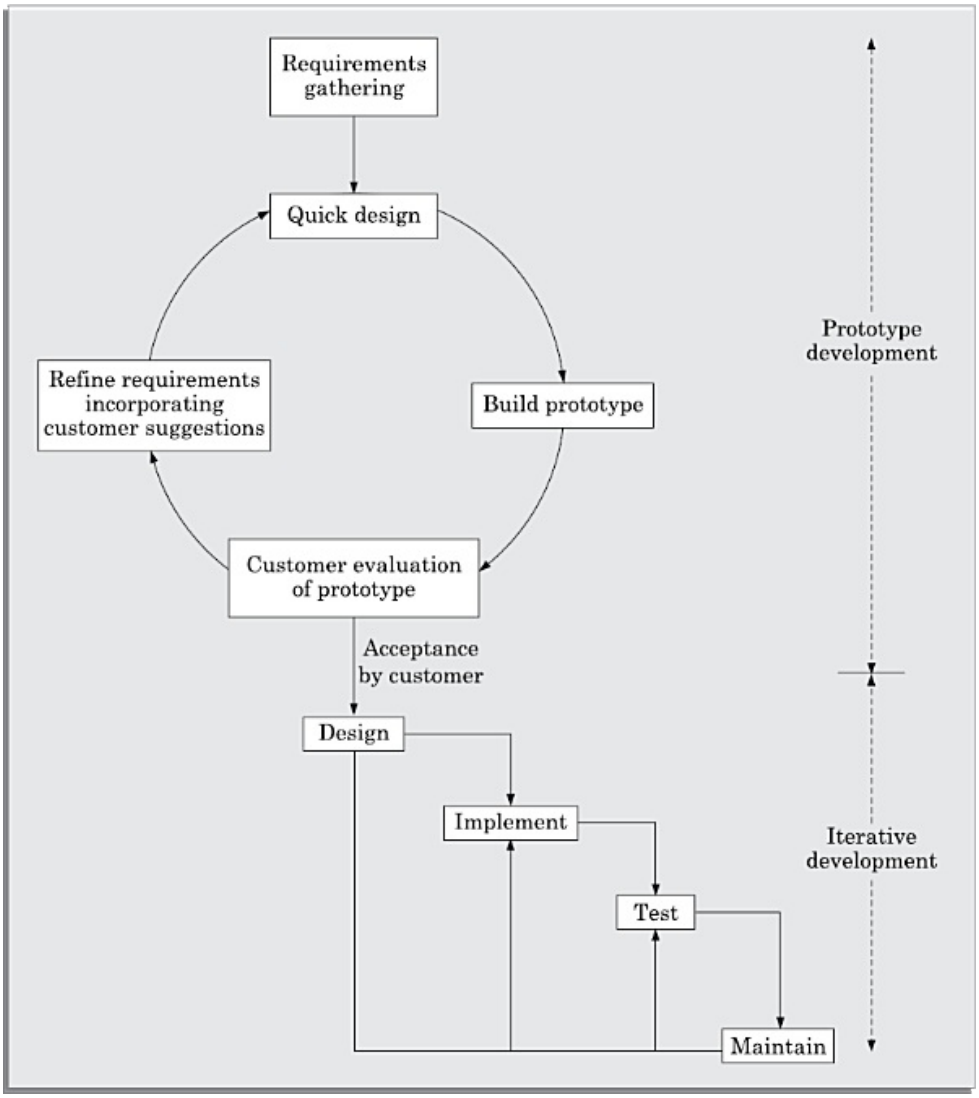


Figure 2.6: Prototyping model of software development.

Even though the construction of a throwaway prototype might involve incurring additional cost, for systems with unclear customer requirements and for systems with unresolved technical issues, the overall development cost usually turns out to be lower compared to an equivalent system developed using the iterative waterfall model.

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

change requests from the customer and the associated redesign costs.

Strengths of the prototyping model

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

Weaknesses of the prototyping model

- The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks.
- 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.

9. EVOLUTIONARY MODEL

- This model has many of the features of the incremental model.
- As in case of the incremental model, the software is developed over a number of increments.
- At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realised.
- In the evolutionary model, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.
- 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 Figure2.9.

Advantages

- **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.

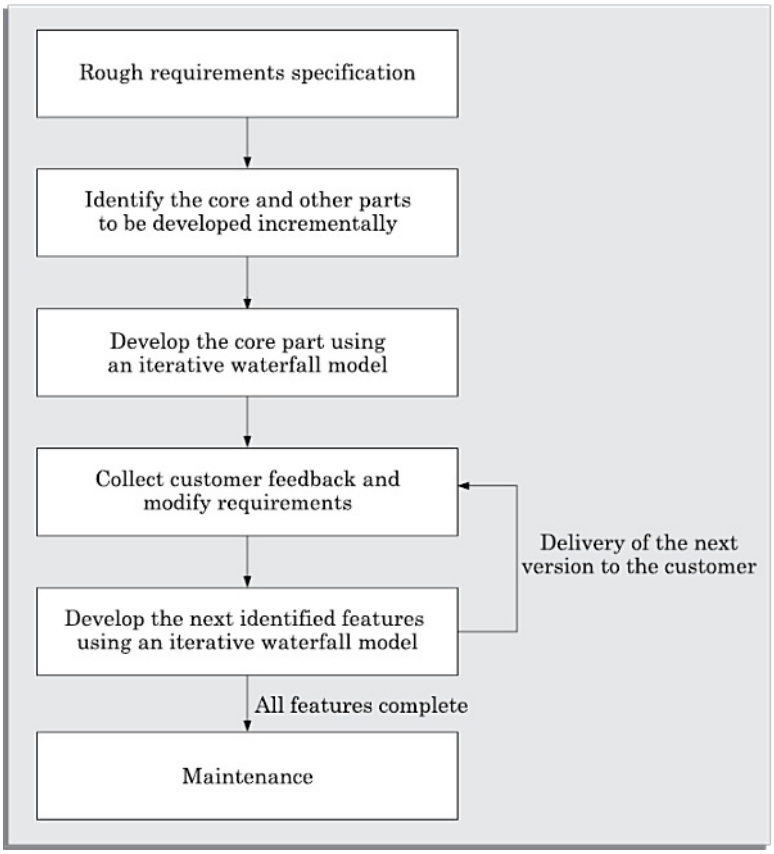


Figure 2.9: Evolutionary model of software development.

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.

Applicability of the evolutionary model

- The evolutionary model is normally useful for very large products, where it is easier to 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.

10. RAPID APPLICATION DEVELOPMENT (RAD)

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

customer feedbacks on incrementally delivered versions.

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

The major goals of the RAD model are as follows:

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

Working of RAD

- In the RAD model, development takes place in a series of short cycles or iterations.
- At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time.
- The time planned for each iteration is called a ***time box***.
- Each iteration is planned to enhance the implemented functionality of the application by only a small amount.
- During each time box, a quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback. Please note that the prototype is not meant to be released to the customer for regular use though.
- The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

How does RAD facilitate accommodation of change requests?

- The customers usually suggest changes to a specific feature only after

they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered.

- Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate faster development?

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

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:** In customised software development projects, substantial reuse is usually made of code from pre-existing software. 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.

- **Highly constrained project 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): T**he 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.

Comparison of RAD with Other Models

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

RAD *versus* prototyping model

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

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

RAD *versus* iterative waterfall model

- In the iterative waterfall model, all the functionalities of a software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse.
- Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that developed using RAD.

RAD *versus* evolutionary model

- Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments

compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

11. 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 fro
- m project to project.
- 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. other models.
- In the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation.
- Over each loop, one or more features of the product are elaborated and analysed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

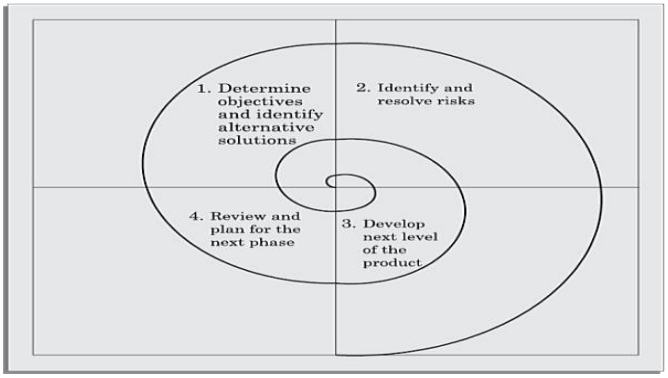


Figure 2.10: Spiral model of software development.

Risk handling in spiral model

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

customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate.

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

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 up 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.

Advantages/pros and disadvantages/cons of the spiral model

- There are a few disadvantages of the spiral model that restrict its use toa 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 this regard, it is much more powerful than the prototyping model.

For projects having many unknown risks that might show up as the development proceeds, the spiral model would be the most appropriate development model to follow.

Prototyping model can meaningfully be used when all the risks associated witha 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).

Unit II

RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

(i) Job Responsibilities for Managing Software Projects

- A software project manager takes the overall responsibility of steering a project to success.
- The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations.
- Most managers take the responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients.
- Classify these activities into two major types—
  - project planning and
  - project monitoring and control.

- **Project planning:**
  - Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.
  - The initial project plans are revised from time to time as the project progresses and more project data become available.
- **Project monitoring and control:**
  - Project monitoring and control activities are undertaken once the development activities start.
  - The focus of project monitoring and control activities is to ensure that the software development proceeds as per plan.

(ii) Skills Necessary for Managing Software Projects

- Three skills that are most critical to successful project management are the following:
  - Knowledge of project management techniques.
  - Decision taking capabilities.

- Previous experience in managing similar projects.

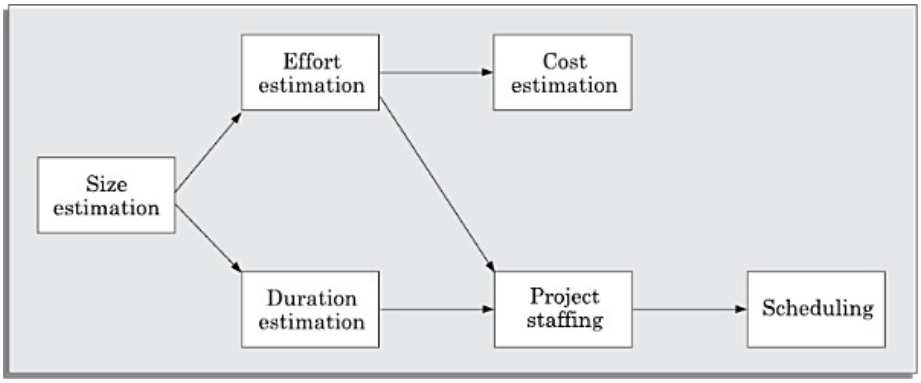
- **Knowledge of project management techniques.**
  - A theoretical knowledge of various project management techniques is certainly important to become a successful project manager.
  - However, a purely theoretical knowledge of various project management techniques would hardly make one a successful project manager.
- **Decision taking capabilities**
  - Effective software project management calls for good qualitative judgment and decision taking capabilities.
  - In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, and configuration management, etc.,
  - Project managers need good communication skills and the ability to get work done.
- **Previous experience in managing similar projects**
  - Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience.

1. PROJECT PLANNING

- Once a project has been found to be feasible, software project managers undertake project planning.
- Project planning is undertaken and completed before any development activity starts.
- For effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial.
- During project planning, the project manager performs the following activities.
  - **Estimation**: The following project attributes are estimated.
  - **Cost**: How much is it going to cost to develop the software product?
  - **Duration**: How long is it going to take to develop the product?
  - **Effort**: How much effort would be necessary to develop the product?

- **Scheduling**: After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.
- **Staffing**: Staff organisation and staffing plans are made.
- **Risk management**: This includes risk identification, analysis, and abatement planning.
- **Miscellaneous plans**: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

➢ Order in which the planning activities are undertaken.



Precedence ordering among planning activities

➢ **Size** is the most fundamental parameter based on which all other estimations and project plans are made.

➢ Based on the size estimation, the effort required to complete a project and the duration over which the development is to be carried out are estimated.

➢ Based on the effort estimation, the cost of the project is computed.

➢ Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made.

(i) Sliding Window Planning

➢ *In the sliding window planning technique, starting with an initial plan, the project is planned more accurately over a number of stages.*

➢ It is usually very difficult to make accurate plans for large projects at project initiation.

➢ **Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as sliding window planning.**

- At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. His information base gradually improves as the project progresses through different development phases.
- The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear.
- The project parameters are re-estimated periodically as understanding grows and also a periodically as project parameters change.
- By taking these developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence.

(ii) The SPMP Document of Project Planning

- Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document.
- **Organisationof the software project management plan (SPMP) document**

1. Introduction
(a) Objectives
(b) Major Functions
(c) Performance Issues
(d) Management and Technical Constraints

2. Project estimates
(a) Historical Data Used
(b) Estimation Techniques Used
(c) Effort, Resource, Cost, and Project Duration Estimates

3. Schedule
(a) Work Breakdown Structure
(b) Task Network Representation
(c) Gantt Chart Representation
(d) PERT Chart Representation

4. Project resources
(a) People
(b) Hardware and Software
(c) Special Resources

5. Staff organisation
(a) Team Structure
(b) Management Reporting

6. Risk management plan
(a) Risk Analysis
(b) Risk Identification
(c) Risk Estimation
(d) Risk Abatement Procedures

7. Project tracking and control plan
(a) Metrics to be tracked

(b) Tracking plan
(c) Control plan

8. Miscellaneous plans

(a) Process Tailoring
(b) Quality Assurance Plan
(c) Configuration Management Plan
(d) Validation and Verification
(e) System Testing Plan
(f) Delivery, Installation, and Maintenance Plan

3. PROJECT ESTIMATION TECHNIQUES

- **Estimation**: It is an attempt to determine how much money, efforts, resources and time it will take to build a specific software based system or project.
- **Who does estimation?**
  - Software manager does estimation using information collected from customers and software Engineers and
- **Steps for Estimations:**
  - Estimate the size of the development product
  - Estimate the effect in person- month or person - hours
- Estimation of various project parameters is a basic project planning activity.
- The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost.
- These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling.
- There are three broad categories of estimation techniques:
  - Empirical estimation techniques
  - Heuristic techniques
  - Analytical estimation techniques

(i) Empirical estimation techniques

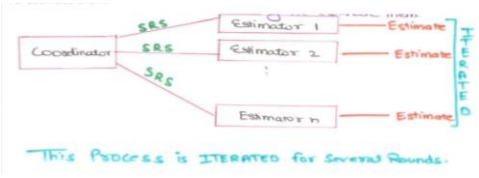
- Empirical estimation techniques are essentially based on making an educated guess of the project parameters.
- While using this technique**, prior experience with development of similar products** is helpful.

- Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalised to a large extent.
- There are two basic empirical estimation techniques are **expert judgement and the Delphi techniques.**
    - **Expert Judgement**
        - Expert judgement is a widely used size estimation technique.
        - In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly.
        - Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate.
        - The outcome of the expert judgement technique is subject to human errors and individual bias.
        - Further, an expert making an estimate may not have relevant experience and knowledge of all aspects of a project.
        - For example, he may be conversant with the database and user interface parts, but may not be very knowledgeable about the computer communication part.
        - Due to these factors, the size estimation arrived at by the judgment of a single expert may be far from being accurate.
        - A more refined form of expert judgement is the estimation made by a group of experts.
        - Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimised when the estimation is done by a group of experts.
        - However, the estimate made by a group of experts may still exhibit bias. For example, on certain issues the entire group of experts may be biased due to reasons such as those arising out of political or social considerations.

- Another important shortcoming of the expert judgement technique is that the decision made by a group may be dominated by overly assertive members.

➢ Delphi Cost Estimation

- Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator.



- In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate.
- Estimators complete their individual estimates anonymously and submit them to the co-ordinator.
- In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations.
- The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators.
- The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate.
- This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process.
- After the completion of several iterations of estimations, the co-ordinator takes the responsibility of compiling the results and preparing the final estimate.
- The Delphi estimation, though consumes more time and effort, overcomes an important shortcoming of the expert judgement technique in that the results can not unjustly be influenced by overly assertive and senior members.

(ii) Heuristic techniques

- Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable **mathematical expressions.**
- Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression.
- Different heuristic estimation models can be divided into the following two broad categories—**single variable and multivariable models.**
    - **S i n g l e variable estimation models**
    - It assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size.
    - A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

      Estimated Parameter = $c1 * e^{d_1}$

        - e - characteristic of the software that has already been estimated (independent variable).
        - Estimated Parameter is the dependent parameter (to be estimated).
        - The dependent parameter to be estimated could be effort, project duration, staff size, etc.,
        - c1 and d1 are constants.
        - The values of the constants c1 and d1 are usually determined using data collected from past projects (historical data).
    - The COCOMO model is an example of a single variable cost estimation model.
    - **Multivariable Cost Estimation**
    - A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter.
    - It takes the following form:

      Estimated Resource = $c1 * p1^{d1} + c2 * p2^{d2} + \ldots$

- where, p1, p2, ... are the basic (independent) characteristics of the software already estimated,
- and c1, c2, d1, d2, are constants.

- Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters.
- The independent parameters influence the dependent parameter to different extents.
- This is modelled by the different sets of constants c1, d1 , c2 , d2 , Values of these constants are usually determined from an analysis of historical data.
- The intermediate COCOMO model to be an example of a multivariable estimation model.

(iii) Analytical estimation techniques

- Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project.
- Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis.
- Halstead's software science is especially useful for estimating software maintenance efforts.
- In fact, it out performs both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned.

4. RISK MANAGEMENT

- ***A risk*** *is any anticipated unfavourable event or circumstance that can occur while a project is underway.*
- Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go hay ware.

- It is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared beforehand to contain each risk.
- In this context, risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real.
- Risk management consists of three essential activities—
    - **risk identification,**
    - **risk assessment, and**
    - **risk mitigation.**

(i) Risk Identification

- The project manager needs to **predict the risks** in a project as early as possible.
- As soon as a risk is identified, **effective risk management plans** are made, so that the possible impacts of the risks is minimised.
- So, early risk identification is important.
- For example, project manager might be worried whether the vendors whom you have asked to develop certain modules might not complete their work in time, whether they would turn in poor quality work, whether some of your key personnel might leave the organisation, etc. All such risks that are likely to affect a project must be identified and listed.
- A project can be subject to a large variety of risks.
- In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes.
- The project manager can then examine which risks from each class are relevant to the project.
- There are three main categories of risks which can affect a software project:
    - **project risks**
    - **technical risks**
    - **business risks**.
- **Project risks**:
    - Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems.
    - An important project risk is schedule slippage.

- Since, software is intangible, it is very difficult to monitor and control a software project.

- **Technical risks:**
  - Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems.
  - Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence.
  - Most technical risks occur due the development team's insufficient knowledge about the product.
- **Business risks:**
  - This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

(ii) Risk Assessment

- The objective of risk assessment is to rank the risks in terms of their damage causing potential.
- For risk assessment, first each risk should be rated in two ways:
  - The likelihood of a risk becoming real (r).
  - The consequence of the problems associated with that risk (s).
- Based on these two factors, the priority of each risk can be computed as follows:

$$p = r \times s$$

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

(iii) Risk Mitigation

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

- There are three main strategies for risk containment:
  - **Avoid the risk:**
    - Risks can be avoided in several ways.
    - Risks often arise due to project constraints and can be avoided by suitably modifying the constraints.
    - The different categories of constraints that usually give rise to risks are:
      - **Process-related risk**: These risks arise due to aggressive work schedule, budget, and resource utilisation.
      - **Product-related risks:** These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability etc.
      - **Technology-related risks:** These risks arise due to commitment to use certain technology (e.g., satellite communication).
    - A few examples of risk avoidance can be the following: Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.
  - **Transfer the risk:**
    - This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.
  - **Risk reduction:**
    - This involves planning ways to contain the damage due to a risk.
    - For example, if there is risk that some key personnel might leave, new recruitment may be planned.
    - The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.
    - For example, if you are using a compiler for recognising user commands, you would have to construct a compiler for a small and very primitive command language first.
- There can be several strategies to cope up with a risk.
- To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk.

- For this we may compute the **risk leverage** of the different risks.
- **Risk leverage** is the difference in risk exposure divided by the cost of reducing the risk. More formally,

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

- Even though we identified three broad ways to handle any risk, effective risk handling cannot be achieved by mechanically following a set procedure, but requires a lot of ingenuity on the part of the project manager.

5. REQUIREMENTS ANALYSIS AND SPECIFICATION

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

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

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

- **System analysts** collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done.
- After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.
- Once the SRS document is ready, it is **first reviewed internally by the project team** to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete.
- The SRS document is then **given to the customer for review**. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.
- Requirements analysis and specification phase mainly involves carrying out the following two important activities:
    - Requirements gathering and analysis
    - Requirements specification

(i) REQUIREMENTS GATHERING AND ANALYSIS

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

(a) 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.
- Requirements gathering may sound like a simple task.
- However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents.

- Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.
- Typically even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed.
- During visit to the customer site, the analysts normally interview the end-users and customer representatives, carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.
- Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete.
- In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:
- **Studying existing documentation:**
    - The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site.
    - Customers usually provide statement of purpose (SoP) document to the developers.
    - Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.
- **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.
    - To systematise this method of requirements gathering, the Delphi technique can be followed.
    - In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This

procedure is repeated till the different users agree on the set of requirements.

- **Task analysis:**
  - The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities).
  - A service supported by a software is also called a task.
  - We can therefore say that the software performs various tasks of the users.
  - In this context, the analyst tries to identify and understand the different tasks to be performed by the software.
  - For each identified task, the analyst tries to formulate the different steps necessary to realise the required functionality in consultation with the users.
- **Scenario analysis:**
  - A task can have many scenarios of operation.
  - The different scenarios of a task may take place when the task is invoked under different situations.
  - For different types of scenarios of a task, the behaviour of the software can be different.
  - For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users. For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.
- **Form analysis:**
  - Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.
  - During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms).
  - In form analysis the exiting forms and the formats of the notifications produced are analysed to determine the data input to the system and the data that are output from the system.

- For the different sets of data input to the system, how these input data would be used by the system to produce the corresponding output data is determined from the users.

(b) Requirements Analysis

- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.
- The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:
  - What is the problem?
  - Why is it important to solve the problem?
  - What exactly are the data input to the system and what exactly are the data output by the system?
  - What are the possible procedures that need to be followed to solve the problem?
  - What are the likely complexities that might arise while solving the problem?
  - If there are external software or hardware with which the developed software has to interface, then what should be the data interchange formats with the external systems?
- After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.
- During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:
  - Anomaly
  - Inconsistency
  - Incompleteness
    - **Anomaly**:
  - It is an anomaly is an ambiguity in a requirement.
  - When a requirement is anomalous, several interpretations of that requirement are possible.

- Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

➢ **Inconsistency:**

- Two requirements are said to be inconsistent, if one of the requirements contradicts the other.

➢ **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 later, possibly while using the software. Often, incompleteness is caused by the inability of the customer to visualise the system that is to be developed and to anticipate all the features that would be required.
- An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

6. SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise the requirements in the form of an SRS document.
- The SRS document usually contains all the user requirements in a structured though an informal form.
- Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write.

(i) Users of SRS Document

- Usually a large number of different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

- **Users, customers, and marketing personnel**:
  - These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs.
  - Remember that the customer may not be the user of the software, but may be some one employed or designated by the user.
  - For generic products, the marketing personnel need to understand therequirements that they can explain to the customers.
- **Software developers:**
  - The software developers refer to the SRS document to make sure that theyare developing exactly what is required by the customer.
- **Test engineers:**
  - The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
- **User documentation writers:**
  - The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- **Project managers:**
  - The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.
- **Maintenance engineers:**
  - The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code.
  - Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose

(ii) Characteristics of a Good SRS Document

- The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects.
- However, the analyst should be aware of the desirable qualities that every good SRS document should possess.
- Some of the identified desirable qualities of an SRS document are the following:
    - **Concise**:
        - The SRS document should be concise and at the same time unambiguous, consistent, and complete.
        - Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
    - **Implementation-independent:**
        - The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
        - It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behaviour of the system and not discuss the implementation issues.
        - The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behaviour of the system. For this reason, **the S R S document is also called the black-box specification** of the software being developed.
        - This view with which a requirements specification is written, has been shown in Figure 4.1. Observe that in Figure 4.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.

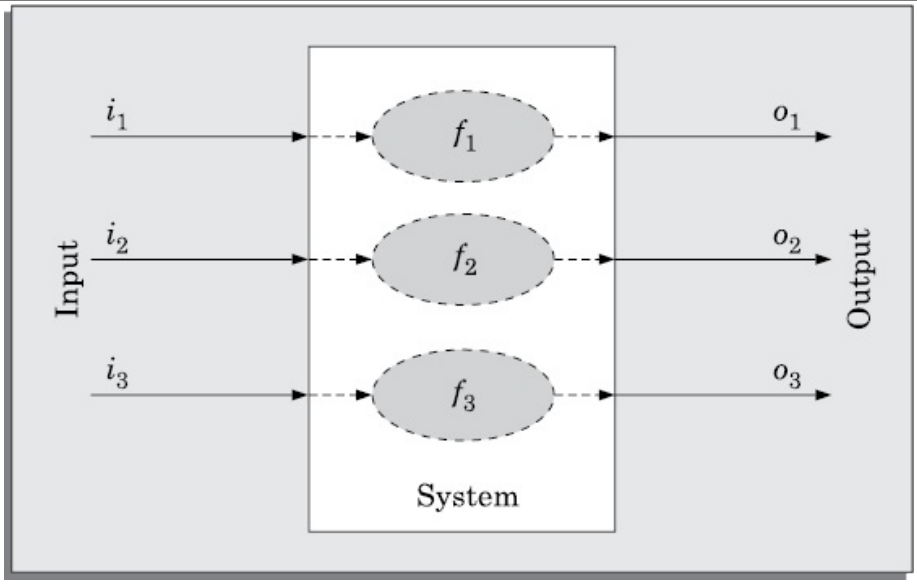


Figure 4.1: The black-box view of a system as performing a set of functions.

- **Traceable**:

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

- **Modifiable**:

- Customers frequently change the requirements during the software development development due to a variety of reasons.
- Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution.
- To cope up with the requirements changes, the SRS document should be easily modifiable.

- For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify.
    - **Identification of response to undesired events:**
- The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.
    - **Verifiable**:
- All requirements of the system as documented in the SRS document should be verifiable.
- This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation.
- A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement—"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable.
- Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

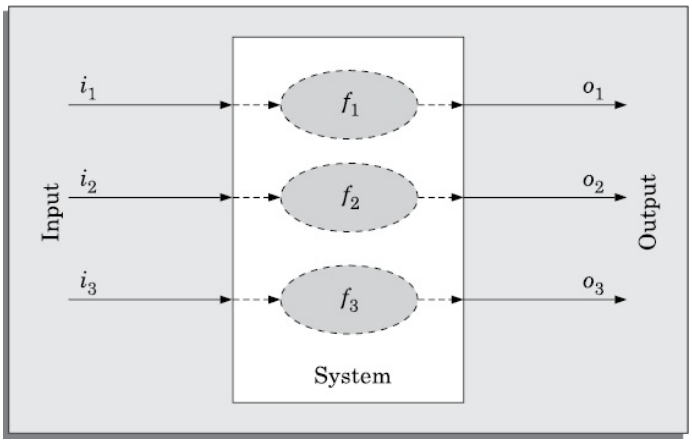
(iii) Important Categories of Customer Requirements

- A good SRS document, should properly categorize and organise the requirements into different sections.
- As per the IEEE 830 guidelines, the important categories of user requirements are the following.
- An SRS document should clearly document the following aspects of a software:
    - **Functional requirements**
    - **Non-functional requirements**
        - **Design and implementation constraints**
        - **External interfaces required**
        - **Other non-functional requirements**
    - **Goals of implementation**.

Functional requirements

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

- To consider a software as offering a set of functions **{fi}** to the user.
- These functions can be considered similar to a mathematical function **f : I → O,** meaning that a function transforms an element **(ii) in the input domain (I) to a value (oi) in the output (O).**
- This functional view of a system is shown schematically in Figure.



- Each function fi of the system can be considered as reading certain data ii, and then transforming a set of input data (ii) to the corresponding set of output data (oi).
- The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set.

Non-functional requirements

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

- The different categories of non- functional requirements that are described under three different sections:
- **Design and implementation constraints:**
  - Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers.
  - Some of the example constraints can be—corporate or regulatory policies that needs to be honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc.
  - Consider an example of a constraint that can be included in this section—**Oracle DBMS** needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.
- **External interfaces required:**
  - Examples of external interfaces are— hardware, software and communication interfaces, user interfaces, report formats, etc.
  - To specify the user interfaces, each interface between the software and the users must be described.
  - The description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on.
  - The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.
- **Other non-functional requirements**:
  - This section contains a description of non- functional requirements that are neither design constraints and nor are external interface requirements.
  - An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in

this section may include reliability issues, accuracy of results, and security issues.

Goals of implementation

- The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed.
- These are not binding on the developers, and they may take these suggestions into account if possible.
- A goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.
- The goals of implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc.
- These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.
- It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and vice versa.

(iv) Functional Requirements

- In order to document the functional requirements of a system, it is necessary to first learn **to identify the high-level functions** of the systems by reading the informal documentation of the gathered requirements.
- The high-level functions would be split into smaller sub 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.

- For example, in a library automation software, a high-level functional requirement might be search-book. This function involves accepting a book name or a set of key words from the user, running a matching algorithm on the book list, and finally outputting the matched books. The generated system response can be in several forms, e.g., display on the terminal, a print out, some data transferred to the other systems, etc. However, in degenerate cases, a high- level requirement may not involve any data input to the system or production of displayable results. For example, it may involve switch on a light, or starting a motor in an embedded application.

- **Are high-level functions of a system similar to mathematical functions?**
  - We all know that a mathematical function transforms input data to output data.
  - A high-level function transforms certain input data to output data.
  - However, 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.
  - In fact, a high-level function usually involves a series of interactions between the system and one or more users.
  - An example of the interactions that may occur in a single high-level requirement has been shown in Figure 4.2.
  - In Figure 4.2, the user inputs have been represented by rectangles and the response produced by the system by circles. Observe that the rectangles and circles alternate in the execution of a single high-level function of the

system, indicating a series of requests from the user and the corresponding responses from the system.

➢

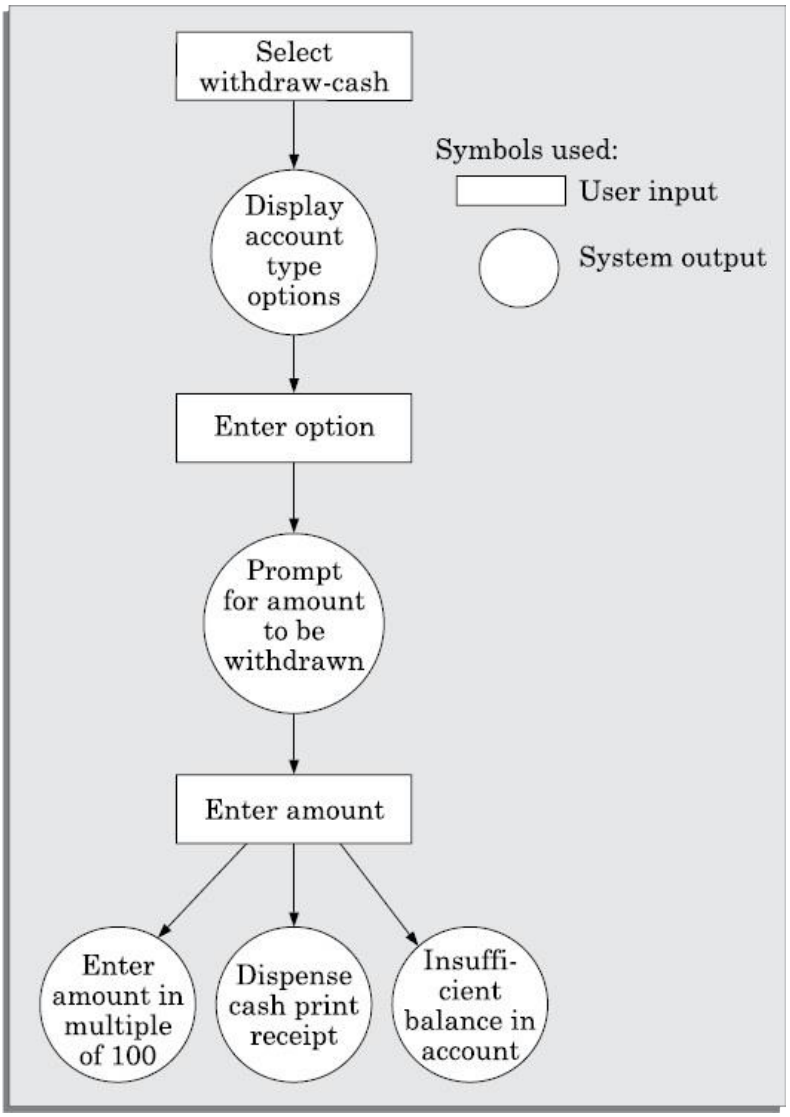


Figure 4.2: User and system interactions in high-level functional requirement.

- Typically , there is some initial data input by the user. After accepting this, the system may display some response (called system action). Based on this, the user may input further data, and so on.
- For any given high-level function, there can be different interaction sequences or scenarios due to users selecting different options or entering different data items.

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

➢ **Is it possible to determine all input and output data precisely?**

- In a requirements specification document, it is desirable to define the precise data input to the system and the precise data output by the system.
- Sometimes, the exact data items may be very difficult to identify.
- This is especially the case, when no working model of the system to be developed exists.
- In such cases, the data in a high-level requirement should be described using high-level terms and it may be very difficult to identify the exact components of this data accurately.
- Another aspect that must be kept in mind is that the data might be input to the system in stages at different points in execution.
- For example, consider the withdraw-cash function of an automated teller machine (ATM) of Figure 4.2. Since during the course of execution of the withdraw-cash function, the user would have to input the type of account, the amount to be withdrawn, it is very difficult to form a single high-level name that would accurately describe both the input data. However, the input data for the subfunctions can be more accurately described.

(v) How to Identify the Functional Requirements?

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

➢ Each high-level requirement characterizes a way of system usage (service invocation) by some user to perform some meaningful piece of work.

➢ **First identify the different types of users who might use the system and thentry to identify the different services expected from the software by differenttypes of users.**

- The decision regarding which functionality of the system can be taken to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some subjectivity.
- For example, consider the issue-book function in a Library Automation System. Suppose, when a user invokes the issue-book function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the issue-book function? Many times, the choice is obvious. But, sometimes it requires making non-trivial decisions.

(vi) Organisation of the SRS Document

- In this section, we discuss the organisation of an SRS document as prescribed by the IEEE 830 standard[IEEE 830].
- Please note that IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it, as may be required for specific projects.
- Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst.
- However, organization of the SRS document to a large extent depends on the preferences of the system analyst himself, and he is often guided in this by the policies and standards being followed by the development company.
- **The introduction section** should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics.
- The introduction section may include the hardware that the system will run on, the devices that the system will interact with and the user skill-levels.
- Description of the user skill-level is important, since the command language design and the presentation styles of the various documents depend to a large extent on the types of the users it is targeted for.
- It is desirable to describe the formats for the input commands, input data, output reports, and if necessary, the modes of interaction.

(a) Introduction

- **Purpose:** This section should describe where the software would be deployed and and how the software would be used.
- **Project scope:** This section should briefly describe the overall context within which the software is being developed. For example, the parts of a problem that are being automated and the parts that would need to be automated during future evolution of the software.
- **Environmental characteristics:** This section should briefly outline the environment (hardware and other software) with which the software will interact.

(b) 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. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.
- **Product features:** This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.
- **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.
- **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. These might include—corporate or regulatory policies; hardware limitations (timing

requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.

- **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

(c) Functional requirements for organisation of SRS document

- This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.

1. User class 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. User class 2
   (a) Functional requirement 2.1
   (b) Functional requirement 2.2

(d) External interface requirements

- **User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard push buttons (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, etc. The details of the user interface design should be documented in a separate user interface specification document.
- **Hardware interfaces:** This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication

protocols to be used.

- **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. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.
- **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. This section should define any pertinent message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any communication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.

(e) Other non-functional requirements for organisation of SRS document

- This section should describe the non-functional requirements other than the design and implementation constraints and the external interface requirements that have been described in Sections 2 and 4 respectively.
- **Performance requirements:** Aspects such as number of transaction to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also 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. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.
- **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the

security issues. Define any security or privacy certifications that must be satisfied.

- For software that have distinct modes of operation, in the functional requirements section, the different modes of operation can be listed and, in each mode, the specific functionalities that are available for invocation can be organised as follows.

Functional requirements

1. Operation mode 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. Operation mode 2
   (a) Functional requirement 2.1
   (b) Functional requirement 2.2

- Specification of the behaviour may not be necessary for all systems.
- It is usually necessary for those systems in which the system behaviour depends on the state in which the system is, and the system transits among a set of states depending on some prespecified conditions and events.
- The behaviour of a system can be specified using either the *finite state machine* (FSM) formalism and any other alternate formalisms. The FSMs can used to specify the possible states (modes) of the system and the transition among these states due to occurrence of events.

Example 4.9 (Personal library software):

Functional requirements

The software needs to support three categories of functionalities as described below:

1. **Manage own books**

Register book

Description: To register a book in the personal library, the details of a book, such as name, year of publication, date of purchase, price and publisher are entered. This is stored in the database and a unique serial number is generated.

Input: Book details

Output: Unique serial number

: Issue book

Description: A friend can be issued book only if he is registered. The various books outstanding against him along with the date borrowed are first displayed.

: Display outstanding books

Description: First a friend's name and the serial number of the book to be issued are entered. Then the books outstanding against the friend should be displayed.

Input: Friend name

Output: List of outstanding books along with the date on which each was borrowed.

: Confirm issue book

If the owner confirms, then the book should be issued to him and the relevant records should be updated.

Input: Owner confirmation for book issue. Output: Confirmation of book issue.

: Query outstanding books

Description: Details of friends who have books outstanding against their name is displayed.

Input: User selection

Output: The display includes the name, address and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued.

: Query book

Description: Any user should be able to query a particular book from anywhere using a web browser.

Input: Name of the book.

Output: Availability of the book and whether the book is issued out.

: Return book

Description: Upon return of a book by a friend, the date of return is stored and the book is removed from the borrowing list of the concerned friend.

Input: Name of the book.

Output: Confirmation message.

2. Manage friend details

: Register friend

Description: A friend must be registered before he can be issued books. After the registration data is entered correctly, the data should be stored and a confirmation message should be displayed.

Input: Friend details including name of the friend, address, land line number and mobile number.

Output: Confirmation of registration status.

: Update friend details

Description: When a friend's registration information changes, the same must be updated in the computer.

: Display current details

Input: Friend name.

Output: Currently stored details.

R2.2.2: Update friend details

Input: Changes needed.

Output: Updated details with confirmation of the changes.

R.3.3: Delete a friend record

Description: Delete records of inactive members.

Input: Friend name.

Output: Confirmation message.

3. Manage borrowed books

: Register borrowed books

Description: The books borrowed by the user of the personal library are registered.

Input: Title of the book and the date borrowed.

Output: Confirmation of the registration status.

: Deregister borrowed books

Description: A borrowed book is deregistered when it is returned.

Input: Book name.

Output: Confirmation of deregistration.

: Display borrowed books

Description: The data about the books borrowed by the owner are displayed.

Input: User selection.

Output: List of books borrowed from other friends.

4. Manage statistics

: Display book count

Description: The total number of books in the personal library should be displayed.

Input: User selection.

Output: Count of books.

R4.2: Display amount invested

Description: The total amount invested in the personal library is displayed.

Input: User selection.

: Display number of transactions *Description:* The total numbers of books issued and returned over a specific period by one (or all) friend(s) is displayed.

Input: Start of period and end of period.

Output: Total number of books issued and total number of books returned.

Non-functional requirements

N.1: Database: A data base management system that is available free of cost in the public domain should be used.

N.2: Platform: Both Windows and Unix versions of the software need to be developed.

N.3: Web-support: It should be possible to invoke the query book functionality from any place by using a web browser.

Observation: Since there are many functional requirements, the requirements have been organised into four sections: Manage own books, manage friends, manage borrowed books, and manage statistics. Now each section has less than 7 functional requirements. This would not only enhance the readability of the document, but would also help in design.

Unit III

OVERVIEW OF THE DESIGN PROCESS

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

(i) Outcome of the Design Process

- The following items are designed and documented during the design phase.
  - **Different modules required:**
    - The different modules in the solution should be clearly identified.
    - Each module is a collection of functions and the data shared by the functions of the module.
    - Each module should accomplish some well-defined task out of the overall responsibility of the software.
    - Each module should be named according to the task it performs.
    - For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.
  - **Control relationships among modules:**
    - A control relationship between two modules essentially arises due to function calls across the two modules.
    - The control relationships existing among various modules should be identified in the design document.
  - **Interfaces among different modules:**
    - The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
  - **Data structures of the individual modules:**
    - Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.
    - Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
  - **Algorithms required to implement the individual modules:**
    - Each function in a module usually performs some processing activity.

- The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due

considerations given to the accuracy of the results, space and time complexities.

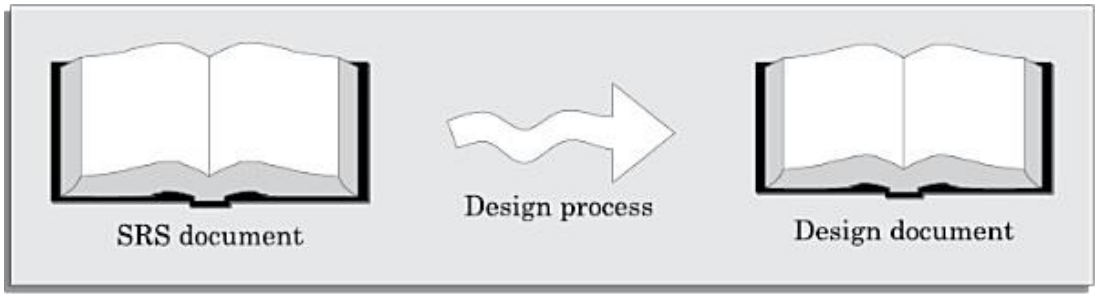


Figure 5.1: The design process

- Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps.
- The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

(ii) Classification of Design Activities

- A good software design is seldom realised by using a single step procedure, rather it requires iterating over a series of steps called the design activities.
- Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.
  - **Preliminary (or high-level) design, and**
  - **Detailed design.**
- The meaning and scope of these two stages can vary considerably from one design methodology to another.

(a) High-Level Design

- However, for the **traditional function-oriented design approach**, it is possible to define the objectives of the high-level design as follows:

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

- The outcome of high-level design is called the **program structure** or **the software architecture.**
- High-level design is a crucial step in the overall design of a software.

- When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.
- Many different types of notations have been used to represent a high-level design.
- A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
- Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design, we confine our attention in this text to structure charts and UML diagrams only.

(b) Detailed Design

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

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

- The outcome of the detailed design stage is usually documented in the form of a

Module Specification (MSPEC) document.

- After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding.

1. HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

- The definition of a "good" software design can vary depending on the exact application being designed.
- For example, "memory size used up by a program" may be an important issue to Characterise a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations.
- Every good software design for general applications must possess some characteristics are listed below:

- **Correctness:**
  - A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability:**
  - A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency:**
  - A good design solution should adequately address resource, time, and cost optimisation issues.
- **Maintainability:**
  - A good design should be easy to change.
  - This is an important requirement, since change requests usually keep coming from the customer even after product release.

(i) Understandability of a Design: A Major Concern

- While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one.
- Obviously all incorrect designs have to be discarded first.
- Out of the correct design solutions, how can we identify the best one?
- Given that we are choosing from only correct design solutions, **understandability of a design solution** is possibly the most important issue to be considered while judging the goodness of a design.
- A good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain.
- A complex design would lead to severely increased life cycle costs.
- Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it.
- About 60 percent of the total effort in the life cycle of a typical product is spent on maintenance. If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold.
- Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable.

- Understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.

An understandable design is modular and layered

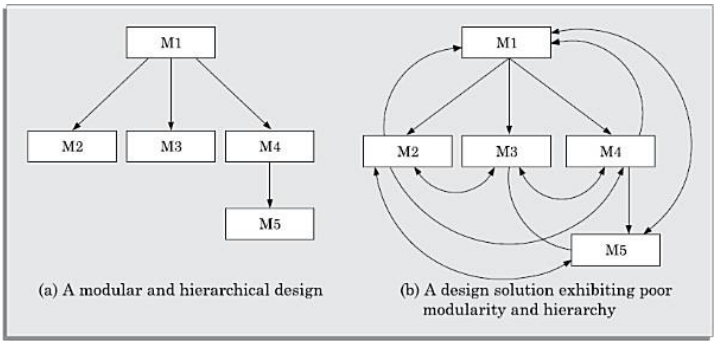
- To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess.
- A design solution should have the following characteristics to be easily understandable:
  - It should assign consistent and meaningful names to various design components.
  - It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
    - **A design solution is understandable, if it is modular and the modulesare arranged in distinct layers**.

- **Modularity**
  - A modular design is an effective decomposition of a problem.
  - It is a basic characteristic of any good design solution.
  - A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.
  - Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.
  - If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly.
    - **How can we compare the modularity of two alternate designsolutions**?
  - For example, consider two alternate design solutions to a problem that are represented in Figure 5.2, in which the modules M1 , M2 etc. have been drawn as rectangles.
  - The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 5.2(a) would

be easier to understand since the interactions among the different modules is low.

- But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another.
- Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterise the modularity of a design solution based on the cohesion and coupling existing in the design.
    - **A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.**
- A software design with high cohesion and low coupling among modules is the effective problem decomposition. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

Figure 5.2: Two design solutions to the same problem.



➢ Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
- In a layered design solution, the modules are arranged in a hierarchy of layers.
- A module can only invoke functions of the modules in the layer immediately below it.

- The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.
- A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.
- A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.
- When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error.
- This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

2. FUNCTION-ORIENTED SOFTWARE DESIGN :

OVERVIEW OF SA/SD METHODOLOGY

- As the name itself implies, SA/SD methodology involves carrying out two distinct activities:
  - Structured analysis (SA)
  - Structured design (SD)
- The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

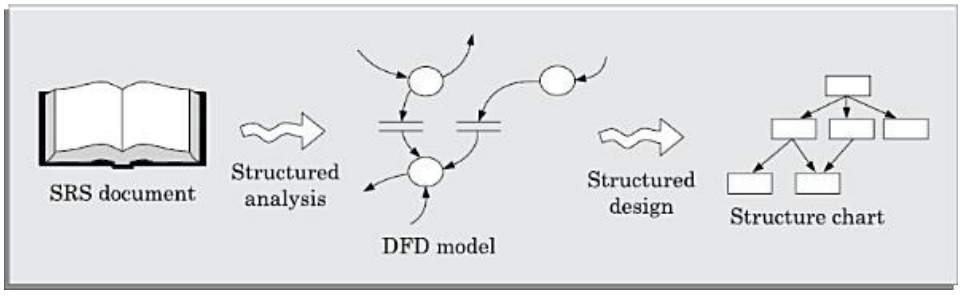


Figure 6.1: Structured analysis and structured design methodology.

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

- During structured design, the DFD model is transformed into a structure chart.

- As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model.
- During structured analysis, functional decomposition of the system is achieved.
- That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions.
- On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure.
- This module structure is also called the high- level design or the software architecture for the given problem. This is represented using a structure chart.
- The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.

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

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

3. STRUCTURED ANALYSIS

- During structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically.
- Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978].
- The structured analysis technique is based on the following underlying principles:

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

- A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
- Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed.
- In the DFD terminology, **each function is called a process or a bubble**.
- It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.
- DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organisation.

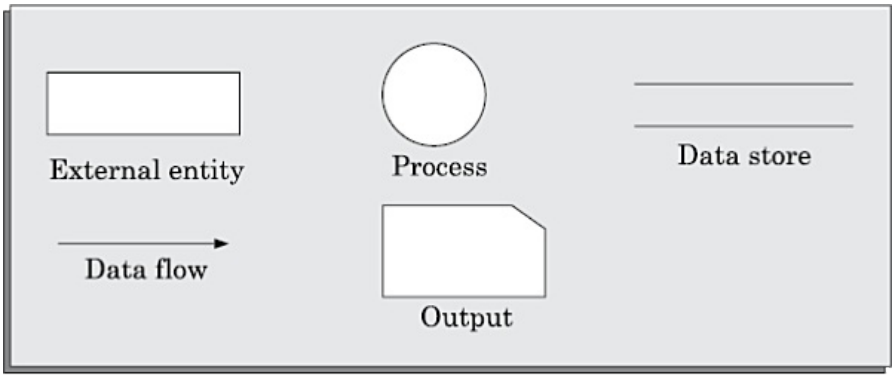
(I) DATA FLOW DIAGRAMS (DFDS)

- 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.
- 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 sub functions performed by the functions using a hierarchy of diagrams.
- The DFD technique is also based on a very simple set of intuitive concepts and rules.
- The different concepts associated with building a DFD model of a system are:

➢ **Primitive symbols used for constructing DFDs**

- There are essentially five different types of symbols used for constructing DFDs.

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.

- **External entity symbol:**

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

- **Data flow symbol:**

- A directed arc (or an arrow) is used as a data flow symbol.
- A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.

- Data flow symbols are usually annotated with the corresponding data names.
- For example the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read-number to validate-number, data- item flowing into read-number, and valid-number flowing out of validate-number.

  - **Data store symbol:**

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

  - **Output symbol:**

- The output symbol is used when a hard copy is produced.

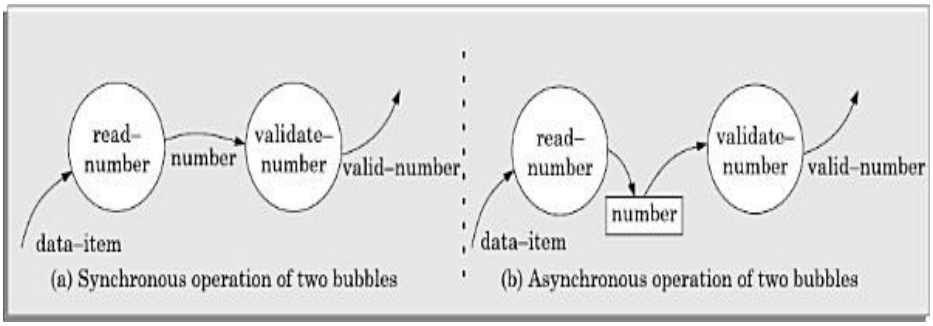
➢ **Important concepts associated with constructing DFD models**

  - **Synchronous and asynchronous operations**

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

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

Figure 6.3: Synchronous and asynchronous data flow.



➢ Data dictionary

- Every DFD model of a system must be accompanied by a data dictionary.
- A data dictionary lists all data items that appear in a DFD model.
- The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model.
- Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs, etc., as shown in Figure 6.4.
- However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.
- A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.
- For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

  grossPay = regularPay + overtimePay
- For the smallest units of data items, the data dictionary simply lists their name and their type.

- Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.
- The dictionary plays a very important role in any software development process, especially for the following reasons:
  - A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
  - The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
  - The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.
- For large systems, the data dictionary can become extremely complex and voluminous.
- Even moderate-sized projects can have thousands of entries in the data dictionary. It becomes extremely difficult to maintain a voluminous dictionary manually.
- Computer-aided software engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary. As a result, the designers do not have to spend almost any effort in creating the data dictionary. These CASE tools also support some query language facility to query about the definition and usage of data items. For example, queries may be formulated to determine which data item affects which processes, or a process affects which data items, or the

definition and usage of specific data items, etc. Query handling is facilitated by storing the data dictionary in a relational database management system (RDBMS).

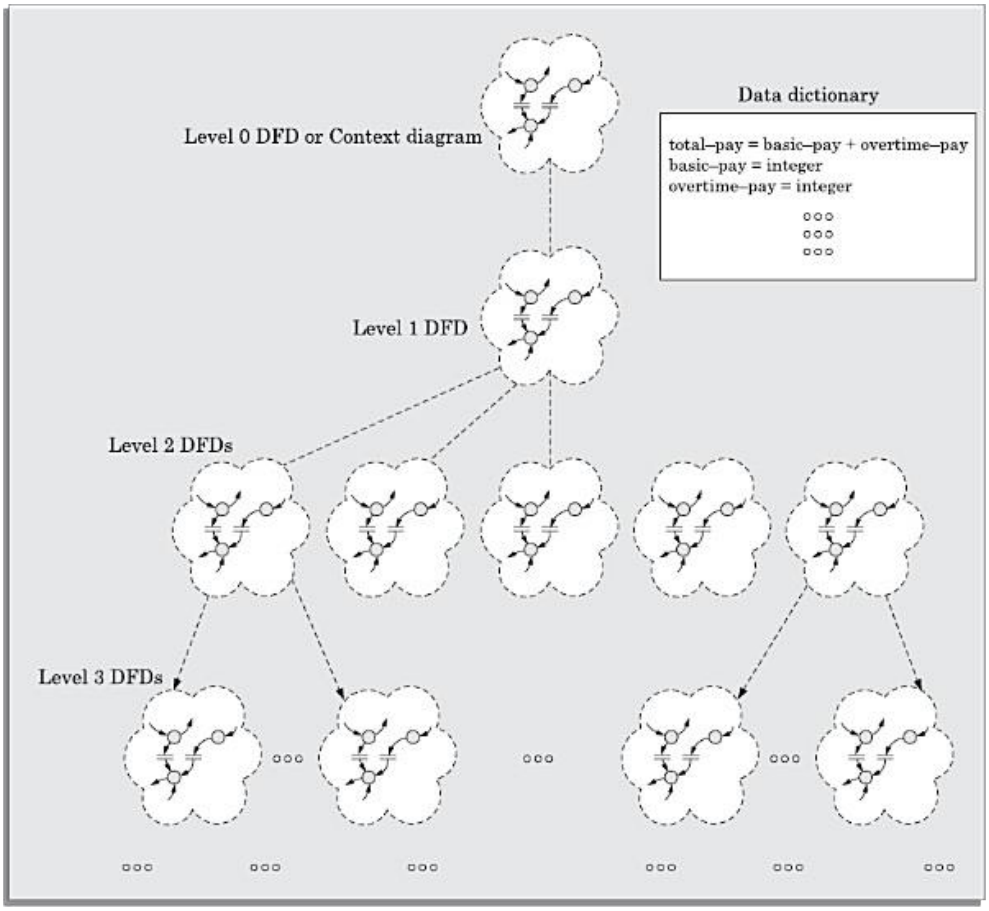
- **Data definition**

  - Composite data items can be defined in terms of primitive data items using the following data definition operators.
    - **+:** denotes composition of two data items, e.g. a+b represents data a and b. [,,]: represents selection, i.e. any one of the data items listed inside the square bracket can occur For example, [a,b] represents either a occurs or b occurs.
    - **():** the contents inside the bracket represent optional data which may or may not appear.
    - **a+(b)** represents either a or a+b occurs.
    - **{}:** represents iterative data definition, e.g. {name}5 represents five name data.
    - **{name}*** represents zero or more instances of name data.
    - **=:** represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.
    - **/* */:** Anything appearing within /* and */ is considered as comment.

4. DEVELOPING THE DFD MODEL OF A SYSTEM

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs
- The DFD model of a problem consists of many of DFDs and a single data dictionary.
- The DFD model of a system is constructed by using a hierarchy of DFDs.
  - **The top level DFD is called the level 0 DFD or the context diagram**.
- This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.

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



(i) 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'.

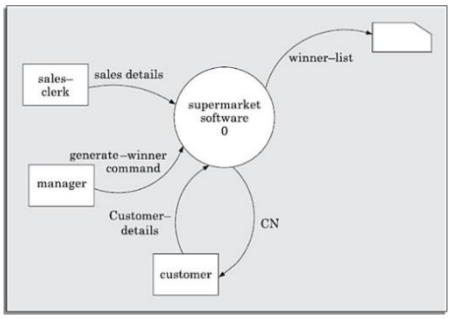


Figure 6.10: Context diagram

- *The context diagram establishes the context in which the system operates; that is,* who are the users, what data do they input to the system, and what data they received by the system.
- The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system.
- The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.
- These data flow arrows should be annotated with the corresponding data names.
- To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users who would be using the system

and the kinds of data they would be inputting to the system and the data theywould 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.
- 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.**

5. STRUCTURED DESIGN

- The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart.
- A structure chart represents the software architecture.
- The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules.
- The basic building blocks using which structure charts are designed are as following:
  - **Rectangular boxes:** A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.
  - **Module invocation arrows:** An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow.
  - **Data flow arrows:** These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the

named data passes from one module to the other in the direction of the arrow.

- **Library modules:** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.
- **Selection:** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol is invoked depending on the outcome of the condition attached with the diamond symbol.
- **Repetition:** A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

- In any structure chart, there should be one and only one module at the top, called the root.
- There should be at most **one control relationship between** any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A.
- Different modules of a structure chart to be arranged in **layers or levels**.
- The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules.
- However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

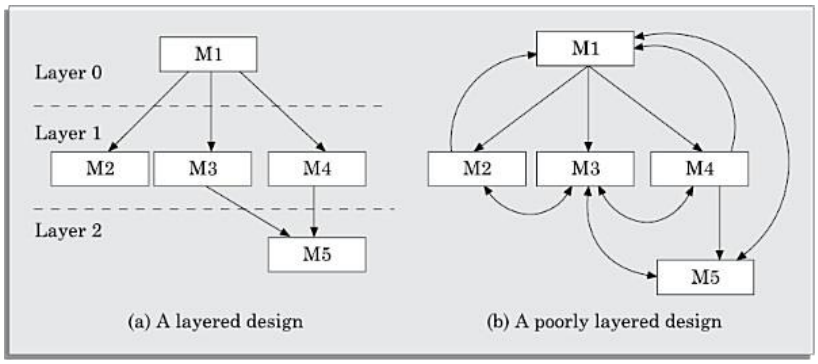


Figure 6.18: Examples of properly and poorly layered designs.

(i) Flow chart versus structure chart

- Flow chart is a convenient technique to represent the flow of control in a program.
- A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

(ii) Transformation of a DFD Model into Structure Chart

- Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart.
- Structured design provides two strategies to guide transformation of a DFD into a structure chart:
  - Transform analysis
  - Transaction analysis
- Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs.
- At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

- To examine the data input to the diagram.
- The data input to the diagram can be easily spotted because they are represented by dangling arrows.
- If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable. Otherwise, transaction analysis is applicable.
- Normally, transform analysis is applicable only to very simple processing.
- Please recollect that the bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code.
- Therefore, transform analysis is normally applicable at the lower levels of a DFD model.

- Each different way in which data is processed corresponds to a separate transaction.
- Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

➢ Transform analysis

- Transform analysis identifies the primary functional components (modules) and the input and output data for these components.
- The first step in transform analysis is to divide the DFD into three types of parts:
    - Input.
    - Processing.
    - Output.
  - The input portion in the DFD includes processes that transform input data from physical (e.g, character from terminal) to logical form (e.g. internal tables, lists, etc.). Each input portion is called an **afferent** branch.
  - The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an **efferent** branch.
  - The remaining portion of a DFD is called central transform.
- In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.
  - Identifying the input and output parts requires experience and skill.
  - One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone.
  - Processes which validate input are not central transforms.
  - Processes which sort input or filter data from it are central transforms.
  - The first level of structure chart is produced by representing each input and output unit as a box and each central transform as a single box.

- In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components.
  - Many levels of functional components may be added.
  - This process of breaking functional components into subcomponents is called factoring.
  - Factoring includes adding read and write modules, error-handling modules, initialisation and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

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

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

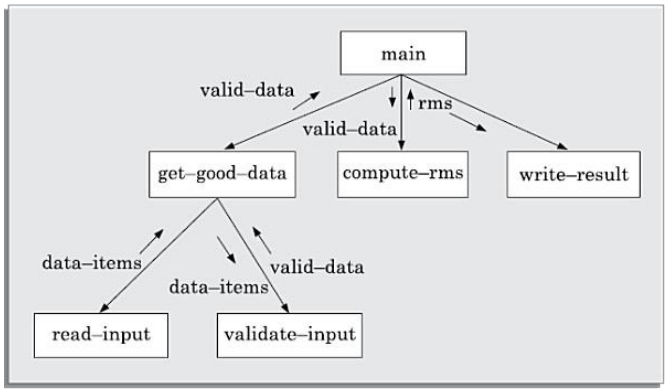


Figure 6.19: Structure chart for Example 6.6.

➢ Transaction analysis

- Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs.
- A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.
- As in transform analysis, first all data entering into the DFD need to be identified.

- In a transaction-driven system, different data items may pass through different computation paths through the DFD.
- This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps.
- Each different way in which input data is processed is a transaction.
- A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.
- For each identified transaction, trace the input data to the output.
- All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart.
- In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type.
- Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.
- Input data to this DFD are handled in three different ways (accept-order, accept- indent-request, and handle-query), we have three different transactions corresponding to these as shown in Figure 6.22.

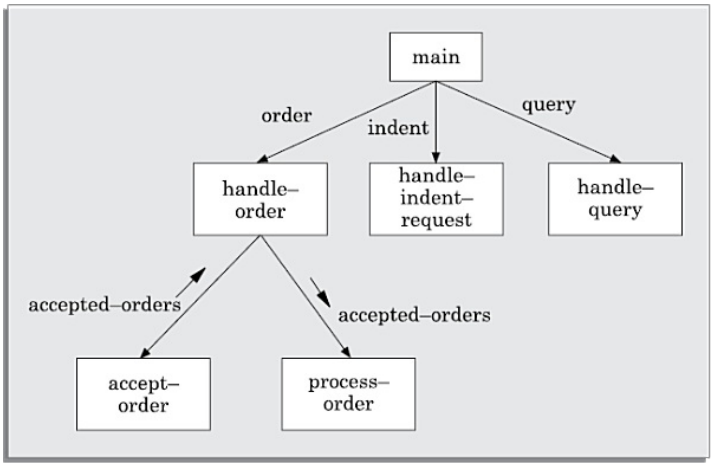


Figure 6.22: Structure chart for Example 6.9.+.

6. DETAILED DESIGN

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.
- These are usually described in the form of module specifications (MSPEC).
- MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower- level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

Unit IV USER INTERFACE DESIGN

1. CHARACTERISTICS OF A GOOD USER INTERFACE

➢ Speed of learning:

- A good user interface should be easy to learn.
- Speed of learning is hampered by complex syntax and semantics of the command issue procedures.
- A good user interface should not require its users to memorise commands.
- Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface.
- Besides, the following three issues are crucial to enhance the speed of learning:

▪ U s e of metaphors and intuitive command names:

- The abstractions of real-life objects or concepts used in user interface design are called metaphors.
- If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it.
- Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

▪ Consistency:

- Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
- This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

▪ Component-based interface:

- Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with.
- This can be achieved if the interfaces of different applications are developed using some standard user interface components.

- The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

➢ Speed of use:

- Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
- This characteristic of the interface is some times referred to as productivity support of the interface.
- It indicates how fast the users can perform their intended tasks.
- The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface.
- For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be

available at the top of a menu to minimise the mouse movements necessary to issue commands.

➢ **Speed of recall:**

- Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximised.
- This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

➢ **Error prevention**:

- A good user interface should minimise the scope of committing errors while initiating different commands.
- The error rate of an interface can be easily determined by monitoring the errors committed by an average users while using the interface.
- This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users.
- Consistency of names, issue procedures, and behaviour of similar commands and the simplicity of the command issue procedures minimise error possibilities. Also, the interface should prevent the user from entering wrong values.

➢ **Aesthetic and attractive:**

- A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

➢ **Consistency:**

- The commands supported by a user interface should be consistent.
- The basic purpose of consistency is to allow users to generalise the knowledge about aspects of the interface from one part to another.
- Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate

➢ **Feedback:**

- A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request.
- In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic.
- If required, the user should be periodically informed about the progress made in processing his command.

➢ **Support for multiple skill levels:**

- A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
- This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.
- Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects.
- Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users.
- When someone uses an application for the first time, his primary concern is speed of learning. After using an application for extended periods of time, he becomes familiar with the operation of the software. As a user becomes more and more familiar with an interface, his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot-keys”, “macros”, etc.
- Thus, the skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

➢ **Error recovery (undo facility):**

- While issuing commands, even the expert users can commit errors.
- Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if

they cannot recover from the errors they commit while using a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

➢ **User guidance and on-line help:**

- Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

2. BASIC CONCEPTS

➢ **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.
    - 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.
- 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

- 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.

3. TYPES OF USER INTERFACES

- User interfaces can be classified into the following three categories:
  - Command language-based interfaces
  - Menu-based interfaces
  - Direct manipulation interfaces

➢ 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.

- Command language-based interfaces are difficult to learn and require the user to memorise the set of primitive commands.
- Also, most users make errors while formulating commands in the command language and also while typing them.
- Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse. Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.
- Issues in designing a command language-based interface Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimise the total typing required.

- We elaborate these considerations in the following:

  - The designer has to decide what mnemonics (command names) to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimise the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.
  - The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development. The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command

composition options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

➢ 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.

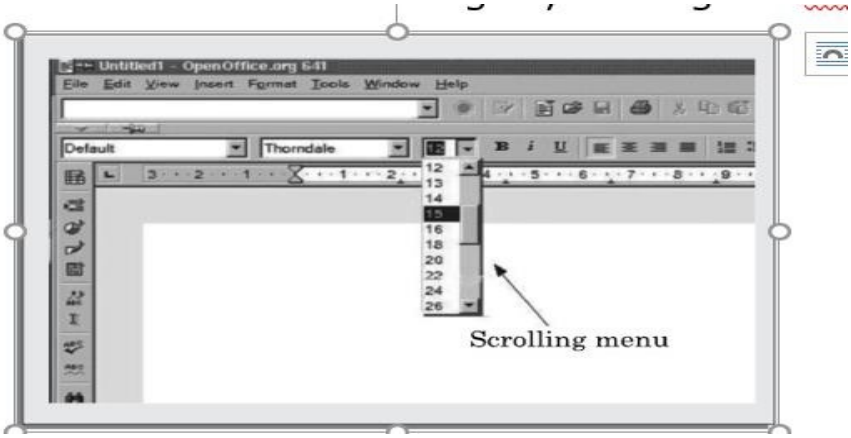


Figure 9.1: Font size selection using scroling menu.

- **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.

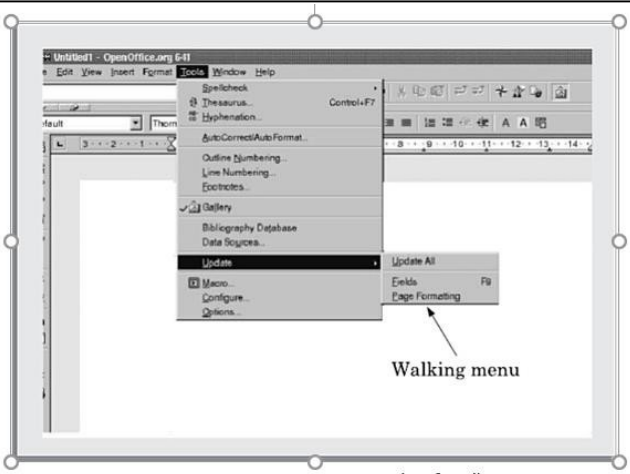


Figure 9.2: Example of walking menu.

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. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files

- —which could be very easily done by issuing a command like delete *.*.

4. FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

Window System

- Most modern graphical user interfaces are developed using some window system.
- A window system can generate displays through a set of windows.
- Since a window is the basic entity in such a graphical user interface, we need to first discuss what exactly a window is.
- **Window**: A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the

user for carrying out independent activities, e.g., one window can be used forediting a program and another for drawing pictures, etc.

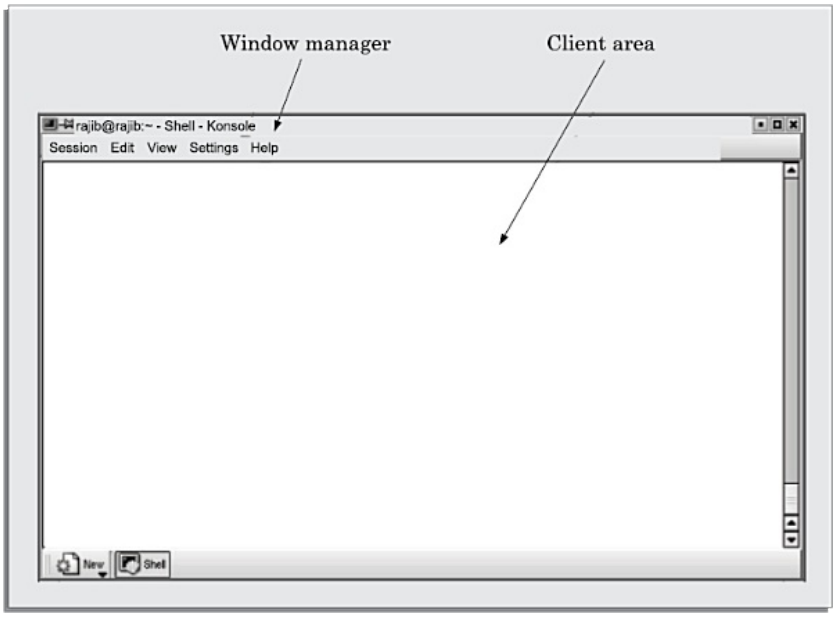


Figure 9.3: Window with client and user areas marked.

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

(i)Window management system (WMS)

- A graphical user interface typically consists of a large number of windows.
- Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS).
- A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use

the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS)

- A WMS simplifies the task of a GUI designer to a great extent by providing the basic behaviour to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application program such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.
- A WMS consists of two parts (see Figure 9.4):
    - a window manager, and
    - a window system.
- These components of the WMS are discussed in the following subsection.
    - **Window manager and window system**:
        - The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system.
        - The window manager and not the window system determines how the windows look and behave.
        - In fact, several kinds of window managers can be developed based on the same window system.
        - Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.
        - The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system.
        - The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure 9.4.
        - This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the

application and the window manger invoke services of the window manager.

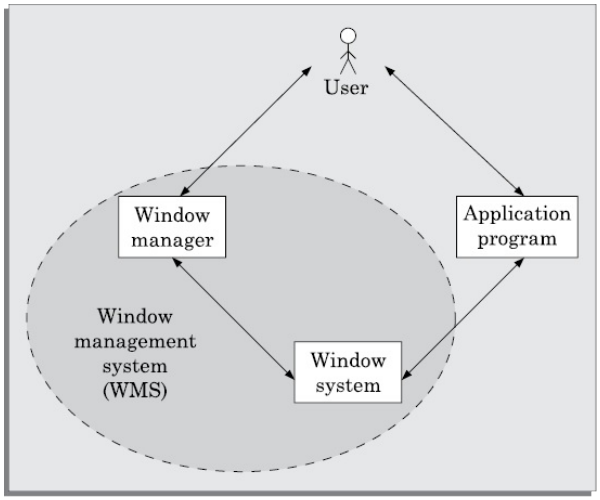


Figure 9.4: Window management system.

- It is usually cumbersome to develop user interfaces using the large set of routines provided by the basic window system.
- Therefore, most user interface development systems usually provide a high-level abstraction called widgets for user interface development.
- A widget is the short form of a window object. We know that an object is essentially a collection of related data with several operations defined on these data which are available externally to operate on these data.
- The data of an window object are the geometric attributes (such as size, location etc.) and other attributes such as its background and foreground colour, etc. The operations that are defined on these data include, resize, move, draw, etc.
- Widgets are the standard user interface components. A user interface is usually made up by integrating several widgets. A few important types of widgets normally provided with a user interface development system are described in Section 9.4.2.

(ii) Component-based development

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

(iii) Visual programming

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

menu bars, icons, and dialog boxes, etc. before adding them to your program. These objects are called as resources. You can design shape, location, type, and size of the dialog boxes before writing any C++ code for the application.

5. CODING

- The input to the coding phase is the design document produced at the end of the design phase.

- During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
  - **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.
- 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.
- 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.

(i) 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.

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.
  - **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.

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

- **Code should be well-documented:**
  - As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.
- **Length of any function should not exceed 10 source lines:**
  - A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
- **Do not use GO TO statements:**
  - Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

6. 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 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.

(i) Internal Documentation

- Internal documentation is the code comprehension features provided in the source code itself.
- Internal documentation can be provided in the code in several forms.
- The important types of internal documentation are the following:
  - Comments embedded in the source code.
  - Use of meaningful variable names.
  - Module and function headers.
  - Code indentation.

- Code structuring (i.e., code decomposed into modules and functions).
- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

- Out of these different types of internal documentation, which one is the most valuable for understanding a piece of code?
  - **Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a pieceof 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.

(ii) 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 required 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.
- 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 of 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

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

- 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.

7. TESTING

BASIC CONCEPTS AND TERMINOLOGIES

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.

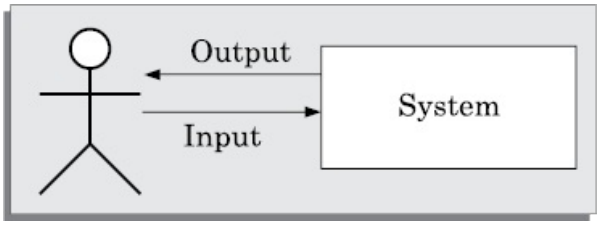


Figure 10.1: A simplified view of program testing

- 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.

Terminologies

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution.
- A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.
- An **error** is the result of a mistake committed by a developer in any of the development activities.
- Among the extremely large variety of errors that can exist in a program.
- One example of an error is a call made to a wrong function.
- The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.
- Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community. Please note that in the domain of hardware testing, the term fault is used with a slightly different connotation [IEEE90] as compared to the terms error and bug.
- **A failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution.
- An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program.
- Every failure is caused by some bugs present in the program.

- The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:
    - The result computed by a program is 0, when the correct result is 10.
    - A program crashes on an input.
    - A robot fails to avoid an obstacle and collides with it.
- It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.
- A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode.
- As an example, consider the different execution modes of a certain text editor software. The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display. In simple words, we can say that a test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.
- A n example of a test case is—[input: “abc”, state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string abc would be displayed.
- **A test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed. An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.
- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.

- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality.
- A test case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system.
- As one example each of a positive test case and a negative test case, consider a program to manage user login. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the login functionality validates and admits a user with wrong or bogus login user name or password.
- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.
- A **failure mode** of a software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.

Verification versus validation

- The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software.
- 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.
- 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 careof 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

8. 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.
- **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.

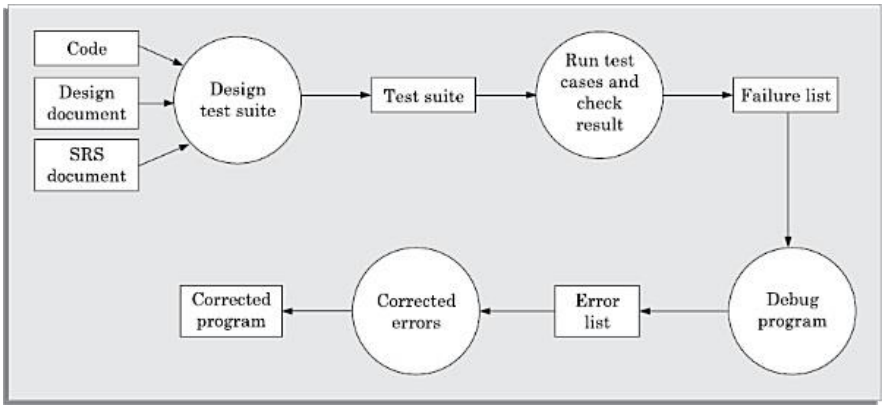


Figure 10.2: Testing process.

➢ 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.

9. 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.
  - **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:**
      - A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour.
      - For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

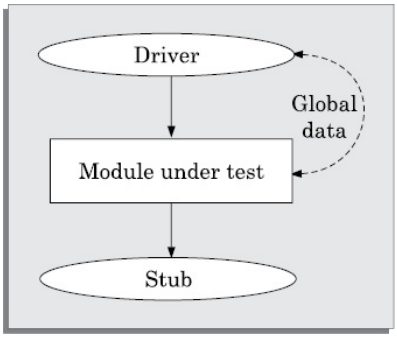


Figure 10.3: Unit testing with the help of driver and stub modules.

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

10. 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

(i) Equivalence Class Partitioning

- In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes.
- The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.
- The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

- Equivalence classes for a unit under test can be designed by examining the input data and output data.
- The following are two general guidelines for designing the equivalence classes:
    - 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,+∞].
    - 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 U-{A,B,C}, where U 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}.

(ii) 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 {0,1,5,10,11}.
- 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,-1,2000,5000,5001}.

11. WHITE BOX TESTING

- White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software rather than just the functionality as in black box testing.
- It is also called glass box testing or clear box testing or structural testing.
  - **Advantages**
    - Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
    - Testing is more thorough, with the possibility of covering most paths.

- **Disadvantages**
  - Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
  - Test script maintenance can be a burden if the implementation changes too frequently.
  - Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

(i) 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.
  - **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.
- **Stronger versus weaker testing**
  - 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.

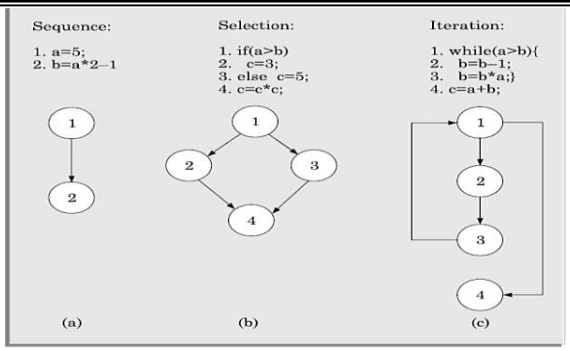


Figure 10.6: I lustration of stronger, weaker, and complementary testing strategies.

(ii) Statement Coverage

- Statement Coverage is a white box testing technique in which all the executable statements in the source code are executed at least once.
- It is used for calculation of the number of statements in source code which have been executed.
- The main purpose of Statement Coverage is to cover all the possible paths, lines and statements in source code.
- Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\textbf{Statement Coverage} = \frac{\textit{Number of executed statements}}{\textit{Total number of statements}} \times 100$$

- **Source Code:**

```
printsum (int a, int b) {---------------------------- Printsum is a function
  int result = a+ b;
  if (result> 0)
        Print ("Positive", result)
  else
        Print ("Negative", result)
  } ------------------------------------ End of the source code
```

- **Scenario 1:**

If A = 3, B = 9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

The statements marked in yellow color are those which are executed as per the scenario

Number of executed statements = 5, Total number of statements = 7

Statement Coverage: 5/7 = 71%

- **Scenario 2:**

If A = -3, B = -9

The statements marked in yellow color are those which are executed as per the scenario.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Number of executed statements = 6

Total number of statements = 7

Statement Coverage: 6/7 = 85%

- But overall if you see, all the statements are being covered by 2nd scenario's considered. So we can conclude that overall statement coverage is 100%.

(iii) Branch Coverage

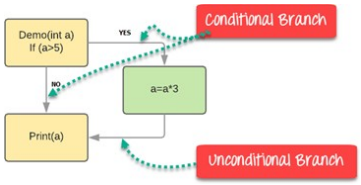
- Branch Coverage is a white box testing method in which every outcome from a code module(statement or loop) is tested.
- The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.
- It helps to measure fractions of independent code segments and to find out sections having no branches.
- If the outcomes are binary, you need to test both True and False outcomes.

- The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

- Example

```
Demo(int a) {
   If (a> 5)
        a=a*3
   Print (a)
  }
```



- Branch Coverage will consider unconditional branch as well

Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	33%
2	6	18	50%	67%

(iv) Condition Coverage

- Also known as **predicate coverage**, it involves testing the condition statement for both True and False values for all the input variables.
- EXAMPLE 5.2 If a and b, then

  Condition Coverage can be satisfied by two tests for True and False values of a andb:

  a = True, b = False

  a = False, b = True

(v) Condition/Branch Coverage

- In this testing, both Condition Coverage and Decision Coverage are covered.

- Every Condition in a decision in the program takes all possible Boolean values at least once, and every Decision in the program takes all possible outcomes at least once.
- Both Decision Coverage and Condition Coverage are satisfied as illustrated in Examples 5.3 and 5.4.
- EXAMPLE 5.3

  If a and b, then

  Condition/Decision coverage can be satisfied by two tests for True and False values of a and b:

  a = True, b = True

  a = False, b = False

```
If(a && b)
{
}
Else
{
}
```

- EXAMPLE 5.4

  If {(a or b) and c}, then

  To satisfy Condition Coverage, each Boolean variable a, b and c in the statementshould be assigned True and False at least one time.

Therefore, the test cases for Condition Coverage are:

Test case #1: a = True, b = True, c = True

Test case #2: a = False, b = False, c = False

To satisfy the Decision Coverage as well, it should be ensured that the IF statementis evaluated to True and False at least once. So the test set will be:

Test case #1: a = True, b = True, c = True

Test case #2: a = False, b = False, c = Falseq

Condition Coverage does not necessarily imply Decision Coverage.

For example, consider the following code:

- EXAMPLE 5.5 If a and b, then

  Condition Coverage can be satisfied by two tests:

a = True, b = False

a = False, b = True

However, this set of tests does not satisfy Decision Coverage as in neithercase, the IF condition will be met for both True and False.

(vi) Multiple Condition Coverage

- Multiple condition coverage checks the True or False outcomes of each condition and requires that all combinations of conditions inside each decision are tested.
- EXAMPLE 5.6 If {(a or b) and c}, then

  It will require eight tests to satisfy Multiple Condition Coverage as thereare 3 variables with 8 combinations:

  - a = False, b = False, c = False
  - a = False, b = False, c = True
  - a = False, b = True, c = False
  - a = False, b = True, c = True
  - a = True, b = False, c = False
  - a = True, b = False, c = True
  - a = True, b = True, c = False
  - a = True, b = True, c = True

  - Test cases are designed such that each component of a condition of a composite conditional expression is given both true and false values.
  - For example, in ((c1 and c2) or c3), each c1, c2, c3 is given True and False values.
  - For a Boolean expression of n variables, 2^n test cases are required.

(vii) Path Coverage

- 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.
- In this type of testing, all paths (which is a sequence of nodes and edges from starting node to terminal node) are executed at least once.
- This is the strongest criteria.

- EXAMPLE 5.8 Consider the CFG shown in Figure 5.2.

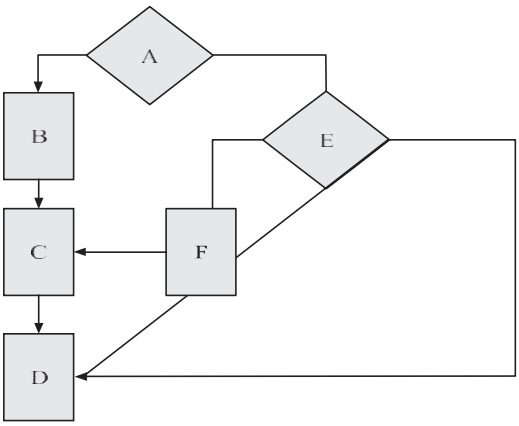


Figure 5.2 Example of CFG.

The graph in Figure 5.2 shows that for statement coverage, nodes A,B,C,D,E,F are to be traversed.

- For Statement Coverage nodes A, B, C, D, E, F are to be traversed.
- For Branch Coverage, links AB BC CD AE EF ED FC are to be traversed.
- For Path Coverage, the paths ABCD, AEFCD, AED are to be traversed.

Control Flow Graph

- A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications.
- Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.
  - **Characteristics of Control Flow Graph:**
    - Control flow graph is process oriented.
    - Control flow graph shows all the paths that can be traversed during a program execution.
    - Control flow graph is a directed graph.
    - Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks.
    - There exist 2 designated blocks in Control Flow Graph:
      - **Entry Block:**

- Entry block allows the control to enter into the control flow graph.

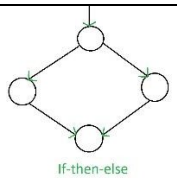
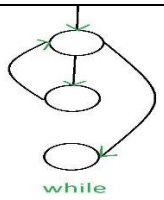
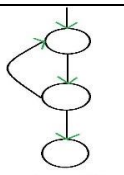
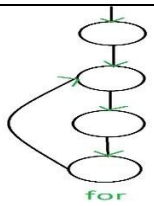
➢ **Exit Block:**

- Control flow leaves through the exit block.

➢ Hence, the control flow graph is comprised of all the building blocks involved in a flow diagram such as the start node, end node and flows between the nodes.

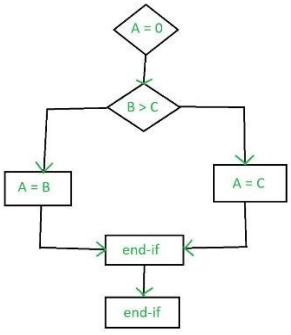
➢ General Control Flow Graphs:

➢ Control Flow Graph is represented differently for all statements and loops. Following images describe it:

If-else:	while:	do-while:	for
			

➢ Example:

```
if A == 10 then
  if B > C
     A = B
  else A = C
  endif
  endif
print A, B, C
```



McCabe Cyclomatic Complexity

- For a control flow graph G of a program, cyclomatic complexity V(G) gives an idea of the number of independent paths. Named after McCabe, who proposed the measure, McCabe cyclomatic complexity defines an upper bound for the number of linearly independent paths and therefore the minimum number of test cases. It can be computed in one of the three ways:
    - **V(G) = E – N + 2,** where N is the number of nodes and E is the number of edges.
    - **V(G) = R,** where R is the number of regions. Area bounded by edges and nodes is called region, and when counting regions, the area outside the graph is counted as a region.
    - **V(G) = P + 1,** where P is the number of binary decision nodes contained in the control flow graph G.
- If a decision is not binary, a three-way decision is counted as two binary decisions. An n-way case statement is counted as n – 1 binary decisions.

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

Uses of McCabe's cyclomatic complexity metric

- **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).

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.

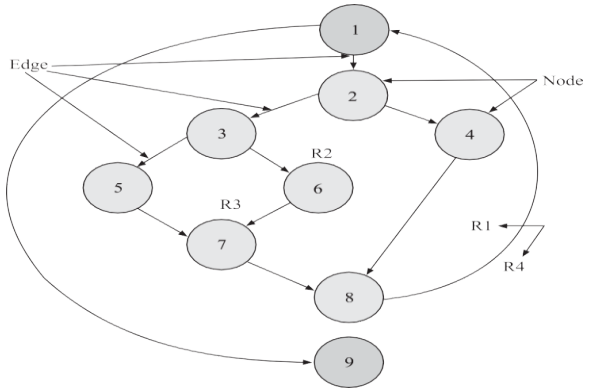
Table 5.1 Guidelines of cyclomatic complexity

➢

Cyclomatic complexity	*Risk factor*
1–10	A simple program without much risk
11–20	More complex, moderate risk
21–50	Complex, high-risk program
>50	Untestable program (very high risk)

➢ Example:

- The calculation of *V*(*G*) is shown in Figure 5.4.



In this example, N = 9, E = 11, R = 4 and P = 3. The cyclomatic complexity V(G) is therefore 4.

(viii) Data Flow Testing

- **Data Flow Testing** is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams. It is concerned with:
    - Statements where variables receive values,
    - Statements where these values are used or referenced.
- To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number.
- For a statement number S-

DEF(S) = {X | statement S contains the definition of X}

USE(S) = {X | statement S contains the use of X}

- If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
- Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables.
- These anomalies are:
    - A variable is defined but not used or referenced,
    - A variable is used but never defined,
    - A variable is defined twice before it is used
        - int a =10
        - int a=5
        - c=a+20

Advantages of Data Flow Testing:

Data Flow Testing is used to find the following issues-

- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.

Disadvantages of Data Flow Testing

- Time consuming and costly process
- Requires knowledge of programming languages

Example:

1. read x, y;

2. if(x>y)

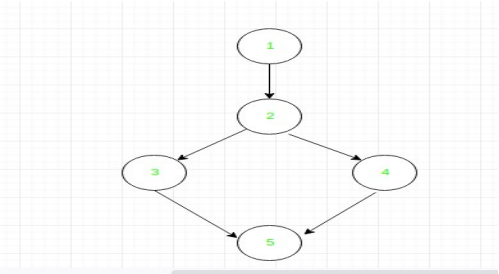
3. a = x+1

else

4. a = y-1

5. print a;

Control flow graph of above example:



Define/use of variables of above example:

VARIABLE	DEFINED AT NODE	USED AT NODE
X	1	2, 3
y	1	2, 4
a	3, 4	5

(ix) Mutation Testing

- Mutation testing evaluates the quality of software tests and does not verify the correctness of the implementation of a given software.
- Mutation testing involves modifying a program's source code in small ways (called mutants) that mimic typical programming errors (such as using the wrong operator or variable name or deleting a statement or introducing an operator), as shown in Figure 4.5.

- It is then checked if the test suite detects the changes in the mutated code. If not, the test suite is considered ineffective.
- The purpose of Mutation Testing is to help the tester develop effective tests or locate weaknesses in the test data used for testing the software.

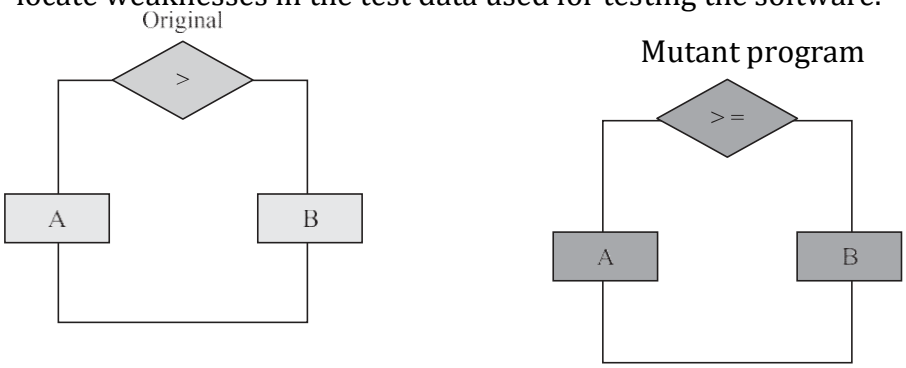


Figure 4.5 Example of creating mutants.

- The following steps are involved in mutation testing (Figure 4.6):

  Step 1: Faults are introduced into the source code of the program and many versions of the program called mutants are created. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail to demonstrate the effectiveness of the test cases.

  Step 2: Test cases are applied to the original program and also to the mutant program. A test case should be adequate and is tweaked to detect faults in a program.

  Step 3: The results of original and mutant program are compared.

  Step 4: (a) If the original program and mutant programs do not generate the same output, it means that the test suite has detected the change. Then that mutant is killed by the test case and the mutant is called mutant killed.

  (b) If the test suite does not detect the mutation, then the mutant is called an equivalent mutant.

  Step 5: If the ratio of all the killed mutants to all the mutants generated is taken, then the value obtained of the metric called Mutation Score gives an idea about quality of the test suite

Mutation Score (MS) = Number of Mutants killed / Total Number of Mutants

- Test suite is Mutation Adequate if its Mutation Score is 1.

Mutants can be created in different ways. For example,

- Each operand can be replaced by every other syntactically legal operand;
- Expressions can be modified by replacing operators and inserting new operators;
- Entire statements can be deleted.

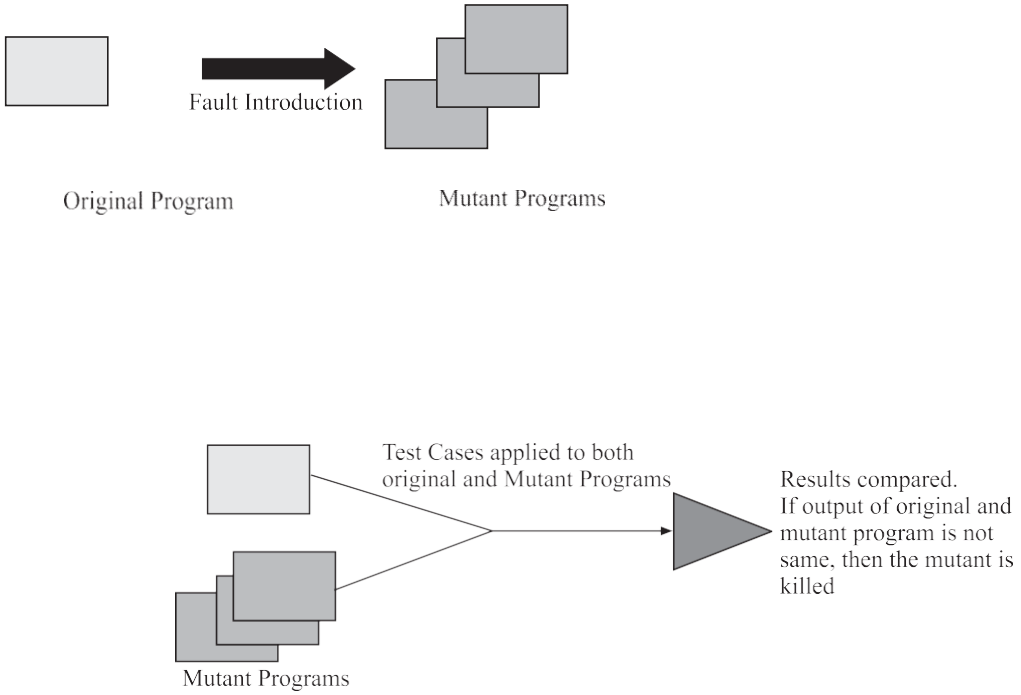


Figure 4.6 Steps in Mutation Testing.

Unit V SOFTWARERELIABILITY

- The reliability of a software product essentially denotes its trustworthiness ordependability.
- Thereliabilityofasoftwareproductcanalsobedefinedastheprobabilityofthe productworking“correctly”overagivenperiodoftime.
- It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced.
- Removingerrorsfromthosepartsofasoftwareproductthatareveryinfrequently executed, makes little difference to the perceived reliability of the product.
- Ithasbeenexperimentallyobservedbyanalysingthebehaviourofalargenumber ofprogramsthat 90percentoftheexecutiontimeofatypicalprogram isspentin executing only 10 percent of the instructions in the program.
- Themostused 10percentinstructionsareoftencalledthe **core1ofa program.**
- Therest90percentoftheprogramstatementsarecalled**non-core**andareonthe average executed only for 10 per cent of the total execution time.
- Itthereforemaynotbeverysurprisingtonote thatremoving60percentproduct defects from the least used parts of a system would typically result in only 3 percent improvement to the product reliability.
- Itisclearthatthequantitybywhichtheoverall reliabilityofaprogramimproves due to the correction ofa single error depends on how frequently the instruction having the error is executed.

- If an error is removed from an instruction that is frequently executed (i.e., belonging to the core of the program), then this would show up as a large improvement to the reliability figure.

- On the other hand, removingerrors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.
- Basedontheabovediscussionwecansaythatreliabilityofaproductdependsnotonly on the number of latent errors but also on the exact location of the errors.
- Apart from this, reliability also depends upon how the product is used, or on its execution profile.
- If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high.
- Ontheotherhand,ifonlythosefunctionsofthesoftwarewhichcontainerrorsare invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low.
- Different categories of users of a software product typically execute different functions of a software product. For example, for a Library Automation Software the library members would use functionalities such as issue book, search book, etc., on the other hand the librarian would normally execute features such as create member, create book record, delete member record, etc.
- So defects which show up for the librarian, may not show up for the members. Suppose the functions of a LibraryAutomationSoftware which the library membersuseareerror-free;andfunctionsusedbytheLibrarianhavemanybugs. Then, these twocategories ofusers would have verydifferent opinions about the reliability of the software.
- Therefore,
- Based on the above discussions, we can summarise the main reasons that make software reliability more difficult to measure than hardware reliability:
- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.

- Theperceivedreliabilityofasoftwareproduct isobserver-dependent.
- Thereliabilityofaproductkeepschangingaserrorsaredetectedandfixed.

- In the following subsection, we shall discuss why software reliability measurement is a harder problem than hardware reliability measurement.

(i)HardwareversusSoftwareReliability

- An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.
- Hardwarecomponentsfailduetoverydifferentreasonsascomparedtosoftware components.
- Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.
- A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part.
- In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug.
- For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors).
- Toput this fact in adifferent perspective, hardware reliabilitystudyisconcerned withstability(for example, the inter-failure times remain constant). Onthe other hand, the aim of software reliability study would be reliability growth (that is, increase in inter-failure times).
- Acomparisonofthechangesinfailurerateovertheproductlifetimeforatypical hardware product as well as a software product are sketched in Figure11.1.

- Observethattheplotofchangeofreliabilitywithtimeforahardwarecomponent (Figure 11.1(a)) appears like a “bath tub”. For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced. The systemthen enters its useful life, where the rate of failure is almost constant. After some time (called product life time) the major componentswearout,andthefailurerateincreases.Theinitialfailuresareusually coveredthroughmanufacturer’swarranty.Acorollaryofthisobservation(though adigressionfromourtopicofdiscussion)isthatitmaybeunwisetobuyaproduct (even at a good discount to its face value) towards the end of its life time, That is, oneneednotfeelhappytobuyatenyearoldcaratonetenthofthepriceofanew car, since it would be near the rising edge of the bath tub curve, and one would havetospendundulylargetime,effort,andmoneyonrepairingandendupasthe loser.
- In contrast to the hardware products, the software product show the highest failure rate just after purchase and installation (see the initial portion of the plot inFigure11.1(b)).Asthesystemisused,moreandmoreerrorsareidentifiedand removed resulting in reduced failure rate. This error removal continues at a slowerpaceduringtheusefullifeoftheproduct.Asthesoftwarebecomesobsolete no more error correction occurs and the failure rate remains unchanged.

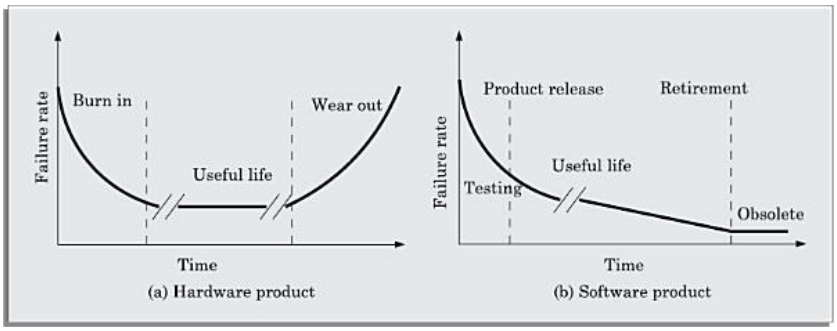


Figure11.1: Changeinfailurerateofaproduct

2.SOFTWAREQUALITY

- Traditionally, the quality of a product is defined in terms of its fitness of purpose.
- Thetraditionalconceptofqualityas“fitnessofpurpose”forsoftwareproducts isnotwholly satisfactory.
- Unlikehardwareproducts,softwarelastsalongtime,inthesensethatitkeeps evolving to accommodate changed circumstances. The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:
    - **Portability**:Asoftwareproductissaidtobeportable,ifitcanbeeasily made to work in different hardware and operating system environments,andeasilyinterfacewithexternalhardwaredevicesand software products.
    - **Usability:**Asoftwareproducthasgoodusability,ifdifferentcategories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability**: A software product is maintainable, if errors can be easilycorrectedasandwhentheyshowup,newfunctionscanbeeasily added to the product, and the functionalities of the product can be easily modified, etc.

➢ **McCall'squalityfactors**

- McCalldistinguishestwolevelsofqualityattributes [McCall].
- The higher- level attributes, known as quality factors or external attributes can only be measured indirectly.
- The second-level quality attributes are called quality criteria. Quality criteria can be measured directly, either objectively or subjectively.
- By combining the ratings of several criteria, we can either obtain a rating for the quality factors, or the extent to which they are satisfied. For example, the reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time.
- Thus, reliability is a higher-level quality factorand number of defects is a low-level quality factor.

➢ **ISO9126**

- ISO9126definesasetofhierarchicalquality characteristics.
- Each sub characteristic in this is related to exactly one quality characteristic. This is in contrast to the McCall's quality attributes that are heavily interrelated. Another difference is that the ISO characteristic strictly refers to a software product, whereas McCall's attributes capture process quality issues as well.

- The users as well as the managers tend to be interested in the higher-level quality attributes (quality factors).

(i) SOFTWAREQUALITYMANAGEMENTSYSTEM

- A quality management system (often referred to as quality system) is the principalmethodologyusedbyorganisationstoensurethattheproductsthey develop have the desired quality.

Managerialstructureandindividualresponsibilities

- Aqualitysystemistheresponsibilityoftheorganisationasa whole.
- However, every organisation has a separate quality department to perform several quality system activities.
- The quality system of an organisation should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

Qualitysystemactivities

- Thequalitysystemactivitiesencompassthefollowing:
  - Auditingofprojectstocheckiftheprocessesarebeing followed.
  - Collectprocessandproductmetricsandanalysethemtocheckif quality goals are being met.
  - Reviewofthequalitysystemtomakeitmoreeffective.
  - Developmentofstandards,procedures,andguidelines.
  - Producereportsforthetopmanagementsummarisingthe effectiveness of the quality system in the organisation.
- Agoodqualitysystemmustbewelldocumented.Withoutaproperly documentedqualitysystem,theapplicationofqualitycontrols and

procedures become ad hoc, resulting in large variations in the quality of the products delivered.

- InternationalstandardssuchasISO9000provideguidance onhowto organise a quality system.

EvolutionofQualitySystems

- Qualitysystemshaverapidlyevolved overthelastsix decades.
- Quality systems of organisations have undergone four stages of evolution as shown in Figure 11.3. The initial product inspection method gave way to quality control (QC) principles.
- Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

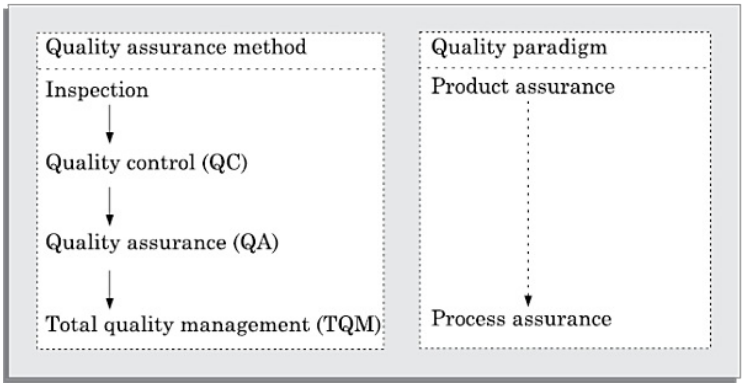


Figure11.3:Evolutionofqualitysystemandcorrespondingshiftinthequality paradigm.

- Thus,qualitycontrolaimsatcorrectingthecausesoferrorsandnotjustrejectingthe defective products. The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.

- The basic premise of modern quality assurance is that if an organisation's processes are good and are followed rigorously, then the products are bound to be of good quality.
- The modern quality assurance paradigm includes guidance for recognising, defining, analysing, and improving the production process.
- Total quality management (TQM) advocates that the process followed by an organisation must continuously be improved through process measurements.
- TQM goes a step further than quality assurance and aims at continuous process improvement.
- TQMgoesbeyonddocumentingprocesses tooptimisingthemthroughredesign.
- Atermrelated toTQMisbusinessprocess re-engineering(BPR),whichisaimsat re-engineering the way business is carried out in an organisation, whereas our focus in this text is re-engineering of the software development process.
- Fromtheabove discussion, we can saythat overthela st sixdecades orso, the quality paradigm has shifted from product assurance to process assurance (see Figure 11.3).

ProductMetricsversusProcessMetrics

- All modern quality systems lay emphasis on collection of certain product and process metrics during product development. Let us first understand the basic differences between product and process metrics.
- Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.
- Examples of product metrics are LOC and function point to measure size, PM (person-month)to measure the effort required to develop it, monthstomeasure the time required to develop the product, time complexity of the algorithms, etc.

- Examples of process metrics are review effectiveness, average number of defects foundperhourofinspection,averagedefectcorrectiontime,productivity,average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

3. ISO9000

- International standards organisation (ISO) is a consortium of 63 countries establishedtoformulateandfosterstandardisation.ISOpublishedits9000series of standards in 1987.

(i) WhatisISO9000Certification?

- ISO 9000 certification serves as a reference for contract between independent parties.
- TheISO9000standardspecifiestheguidelinesformaintainingaqualitysystem.
- We have already seen that the quality system of an organisation applies to all itsactivities related to its products or services.
- The ISO standard addresses both operational aspects (that is, the process) and organisational aspects such as responsibilities, reporting, etc.
- ISO 9000 specifies a set of recommendations for repeatable and high quality product development.
- It is important to realise that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product it self.
- TheISO9000seriesofstandardsarebasedonthepremisethatifaproperprocess is followed for production, then good quality products are bound tofollow automatically.
- ISO9000isaseriesofthreestandards—ISO9001,ISO9002,andISO 9003.

- Thetypesofsoftwarecompaniestowhichthedifferent ISOstandardsapplyareas follows:
- **ISO 9001:** This standard applies to the organisations engaged in design,development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.
- **ISO 9002:** This standard applies to those organisations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organisations.
- **ISO9003:**Thisstandardappliestoorganisationsinvolvedonlyininstallationandtesting of products.

(ii) ISO9000forSoftwareIndustry

- ISO 9000 is a generic standard that is applicable to a large range of industries, starting from a steel manufacturing industry to a service rendering company.
- Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organisations.
- Animportantreasonbehindsuchasituationisthefactthatsoftwaredevelopment is in many respects radically different from the development of other types of products.
- Twomajordifferencesbetweensoftwaredevelopmentanddevelopmentofother kinds of products are as follows:
    - Softwareisintangibleand thereforedifficulttocontrol.

- It means that software would not be visible to the user until the development is complete and the software is up and running.
- Itisdifficulttocontrolandmanageanythingthatyoucannotseeandfeel.
- In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to accurately determine how much work has been completed and to estimate how much more time will it take.
- Duringsoftwaredevelopment,theonlyrawmaterialconsumedis data.
- In contrast, large quantities of raw materials are consumed during the developmentofanyotherproduct.Asanexample,considerasteelmaking company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc.
- Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organisations.

- Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry.
- Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry.
- Atpresent,officialguidanceisinadequateregardingtheinterpretationofvarious clauses of ISO 9000 standard in the context of software industry and one has to keep on cross referencing the ISO 9000-3 document.

(iii) ShortcomingsofISO9000Certification

- Even though ISO 9000 is widely being used for setting up an effective quality system in an organisation, it suffers from several shortcomings.
- SomeoftheseshortcomingoftheISO9000certificationprocessarethefollowing:
  - ISO9000requiresasoftwareproductionprocesstobeadheredto,butdoes not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
  - ISO 9000 certification process is not fool-proof and no international accreditionagencyexists.Thereforeitislikelythatvariationsinthenorms ofawardingcertificatescanexistamongthedifferentaccreditionagencies and also among the registrars.
  - Organisations getting ISO 9000 certification often tend to downplay domainexpertiseandtheingenuityofthedevelopers.Theseorganisations start to believe that since a good process is in place, the development resultsaretrulyperson-independent. Thatis, anydeveloperisaseffective asanyotherdeveloperinperforminganyparticularsoftwaredevelopment activity. In manufacturing industry there is a clear link between process qualityandproductquality.Onceaprocessiscalibrated,itcanberunagain and again producing quality goods. Many areas of software development are so specialised that special expertise and experience in these areas (domain expertise) is required. Also, unlike in case of general product manufacturing, ingenuity and effectiveness of personal practices play an importantpart in determining the results produced by a developer. In other words, software development is a creative process and individual skills and experience are important.
  - ISO9000doesnotautomaticallyleadtocontinuousprocessimprovement. In other words, it does not automatically lead to TQM.

4. SEI CAPABILITY MATURITY MODEL

- SEI capability maturity model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA.
- TheUnitesStatesDepartmentofDefence(USDoD)isthelargestbuyerofsoftware product.Itoftenfaceddifficultiesinvendorperformances,andhadtomanytimes live with low quality products, late delivery, and cost escalations. In this context, SEICMMwasoriginallydevelopedtoassisttheU.S.DepartmentofDefense(DoD) in software acquisition.
- The rationale was to include the likely contractor performance as a factor in contract awards. Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts.
- It was observed that the SEI CMM model helped organisations to improve the qualityofthe software theydeveloped and therefore adoption ofSEICMM model had significant business benefits. Gradually many commercial organisations began to adopt CMM as a framework for their own internal improvement initiatives.
- In simple words, CMM is a reference model for apprising the software process maturityintodifferentlevels.Thiscanbeusedtopredictthemostlikelyoutcome to be expected from the next project that the organisation undertakes.
- It must be remembered that SEI CMM can be used in two ways—capabilityevaluation and software process assessment.
- Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result.
- Capability evaluation provides a way to assess the software process capability of an organisation. Capability evaluation is administered by the contract awarding

authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work. On the other hand, software process assessment is used by an organisation with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.

- ThedifferentlevelsofSEICMMhavebeendesignedsothatitiseasyforan organisation to slowly build its quality system starting from scratch.
- SEICMMclassifiessoftwaredevelopmentindustriesintothefollowing**fivematurity levels:**
- **Level1:Initial**
  - A software development organisation at this level is characterised by adhoc activities. Very few or no processes are defined and followed.
  - Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level.
  - Thesuccessofprojectsdependonindividualeffortsand heroics.
  - Whenadeveloperleavestheorganisation,thesuccessorwouldhavegreat difficulty in understanding the process that was followed and the work completed.Also,noformalprojectmanagementpracticesarefollowed.As a result, time pressurebuilds up towards theend ofthe deliverytime,as a result short-cuts are tried out leading to low quality products.
- **Level2:Repeatable**
  - Atthislevel,thebasicprojectmanagementpracticessuch**astrackingcost and schedule are established**.
  - Configuration management tools are used on items identified for configuration control.

- Sizeandcostestimationtechniquessuchasfunctionpointanalysis,COCOMO, etc., are used.
- The necessary process discipline is in place to repeat earlier success onprojects with similar applications.
- Though there is a rough understanding among the developers about theprocess being followed, the process is not documented.
- Configurationmanagementpracticesareusedforallprojectdeliverables.
- Please remember that opportunity to repeat a process exists only when a company produces a family of products. Since the products are very similar, the success storyon development ofone product can repeatedfor another.
- In a non- repeatable software development organisation, a software product development project becomes successful primarily due to the initiative,effort,brilliance,orenthusiasmdisplayedbycertainindividuals.
- On the other hand, in a non-repeatable software development organisation, the chances of successful completion of a software project is to a great extent depends on who the team members are. For this reason, the successful development of one product by such an organisation does not automatically imply that the next product development will be successful.

- **Level3:Defined**
  - Atthislevel,theprocessesfor**bothmanagementanddevelopment activities are defined and documented**.
  - Thereisacommonorganisation-wideunderstandingofactivities,roles, and responsibilities.
  - The processes though defined, the process and product qualities are not measured.

- At this level, the organisation builds up the capabilities of its employees through periodic training programs. Also, review techniques are emphasized and documented to achieve phase containment of errors.
- ISO9000aimsatachievingthis level.

- **Level4:Managed**
  - Atthislevel, thefocusison**softwaremetrics**.
  - Bothprocessandproductmetricsarecollected.
  - Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met.
  - Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality.
  - Theprocessmetricsareusedtocheckifaprojectperformedsatisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.
- **Level5:Optimising**
  - Atthisstage,**processandproductmetricsare collected**.
  - Process and product measurement data are analysed for continuous process improvement.
  - For example, if from an analysis of the process measurement results, it is found that the code reviews are not very effective and a large number of errorsaredetectedonlyduringtheunittesting,thentheprocesswouldbe fine tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated into the process.
  - Continuous process improvement is achieved both by carefully analysing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies.

- At CMM level 5, an organisation would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organisation- wide.
- Level5organisationsusuallyhaveadepartmentwhosesoleresponsibility is to assimilate latest tools and technologies and propagate them organisation-wide. Since the process changes continuously, it becomes necessary to effectively manage a changing process.
- Therefore,level5organisationsuseconfigurationmanagementtechniques to manage process changes.

- Exceptforlevel1,eachmaturitylevelischaracterisedbyseveralkeyprocessareas (KPAs) that indicate the areas an organisation should focus to improve its software process to this level from the previous level.
- EachofthefocusareasidentifiesanumberofkeypracticesoractivitiesthatneedtobeimplementedTable11.1FocusareasofCMMlevelsandKeyProcessAreas.
- Inotherwords,KPAscapturethefocusareasofalevel.Thefocusofeachleveland the corresponding key process areas are shown in the Table 11.1:

Table11.1FocusareasofCMMlevelsandKeyProcessAreas

Table11.1 FocusareasofCMMlevelsandKeyProcessAreas

CMMLevel	*Focus*	*KeyProcessAreas(KPAs)*
Initial	Competentpeople	
Repeatable	Project management	Software project planning Software configuration management
Defined	Definition of processe s	Process definition Training programPeer reviews
Managed	Product and process quality	Quantitativeprocess metrics Software quality management

Optimising	Continuous process improvement	Defectprevention Process change management Technology change management

- SEICMMprovidesalistofkeyareasonwhichtofocustotakeanorganisationfrom one level of maturity to the next.
- Thus,itprovidesawayforgradualqualityimprovementoverseveral stages.
- Eachstagehasbeencarefullydesignedsuchthatonestageenhancesthecapability already built up. For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.
- SubstantialevidencehasnowbeenaccumulatedwhichindicatethatadoptingSEI CMM has several business benefits. However, the organisations trying out the CMM frequently face a problem that stems from the characteristic of the CMM itself.
- **CMMShortcomings**:CMMdoessufferfromseveralshortcomings.Theimportant among these are the following:
    - ThemostfrequentcomplaintbyorganisationswhiletryingouttheCMM-basedprocessimprovementinitiativeisthattheyunderstandwhatis neededtobeimproved,buttheyneedmoreguidanceabouthowtoimprove it.
    - Anothershortcoming(thatiscommontoISO9000)isthatthicker documents, more detailed information, and longer meetings are considered to be better. This is in contrast to the principles of software economics—reducing complexity and keeping the documentation to the minimum without sacrificing the relevant details.
        - Gettinganaccuratemeasureofanorganisation'scurrentmaturitylevelis alsoanissue.TheCMMtakesanactivity-basedapproachtomeasuring

maturity;ifyoudotheprescribed setofactivitiesthenyouareatacertain level. There is nothing that characterises or quantifies whether you do these activities well enough to deliver the intended results.

5. SOFTWAREMAINTENANCE

- Software maintenance denotes any changes made to a software product after it has been delivered to the customer.
- Maintenanceisinevitableforalmostanykindofproduct.However,mostproducts need maintenance dueto the wear and tear caused by use.
- Needmaintenancetocorrecterrors,enhancefeatures,porttonewplatforms, etc.

(i) CHARACTERISTICSOFSOFTWAREMAINTENANCE

- 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.
- **TypesofSoftware Maintenance**
- Therearethreetypesofsoftwaremaintenance,whicharedescribed asfollows:
- **Corrective:**
    - Correctivemaintenanceofasoftwareproductisnecessaryeithertorectify the bugs observed while the system is in use.

- **Adaptive:**
  - Asoftwareproductmightneedmaintenancewhenthecustomersneedthe producttorunonnewplatforms,onnewoperatingsystems,orwhenthey 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.

(ii) CharacteristicsofSoftwareEvolution

- LehmanandBeladyhavestudiedthecharacteristicsofevolutionofseveral software products [1980].
- Theyhaveexpressedtheirobservationsintheformof laws.
- These are generalisations 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'sfirstlaw:**
  - A software product must change continually or become progressively less useful.
  - Everysoftwareproductcontinuestoevolveafteritsdevelopmentthrough maintenance efforts.
  - Larger products stay in operation for longer times because of higherreplacement costsandthereforetendtoincurhighermaintenanceefforts.
  - This law clearly shows that every product irrespective of how well designedmustundergomaintenance.Infact,whenaproductdoesnotneed any more maintenance, it is a sign that the product is about to be retired/discarded.Thisisincontrasttothecommonintuitionthatonly

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

- **Lehman'ssecondlaw:**
  - The structure of a program tends to degrade as more and moremaintenance is carried out on it.
  - The reason for the degraded structure is that when you add a function duringmaintenance,youbuildontopofanexistingprogram,ofteninaway 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 documentationsbecomeinconsistentandbecomelesshelpfulasmoreand more maintenance is carried out.
- **Lehman'sthirdlaw:**
  - Over a program's lifetime, its rate of development is approximatelyconstant.Therateofdevelopmentcanbequantifiedintermsoft helinesof code written or modified. Therefore this law states that therate at which codeiswrittenormodifiedisapproximatelythesameduringdevelopment and maintenance.

(iii) SpecialProblemsAssociatedwithSoftwareMaintenance

- Softwaremaintenanceworkcurrentlyistypicallymuchmoreexpensivethanwhat itshouldbeandtakesmoretimethanrequired.Thereasonsforthissituationare the following:
- Software maintenance work in organisations is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering.

- Software maintenance has a very poor image in industry. Therefore, an organisation often cannot employ bright engineers to carry out maintenance work.
- Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products. Though the word legacy implies "aged" software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain. The typical problem associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lackofpersonnelknowledgeableintheproduct.Manyofthelegacysystemswere developed long time back. But it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

6. SOFTWARER EVERSEENGINEERING

- 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 understand ability, without changing any of its functionalities.
- Away to carry out these cosmetic changes is shown schematically in Figure13.1.

- A program can be reformatted using any of the several available pretty printer programs which layout the program neatly.
- Manylegacysoftware products aredifficulttocomprehend withcomplexcontrol structureandunthoughtfulvariablenames.Assigningmeaningfulvariablenames is important because that meaningful variable names is 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.

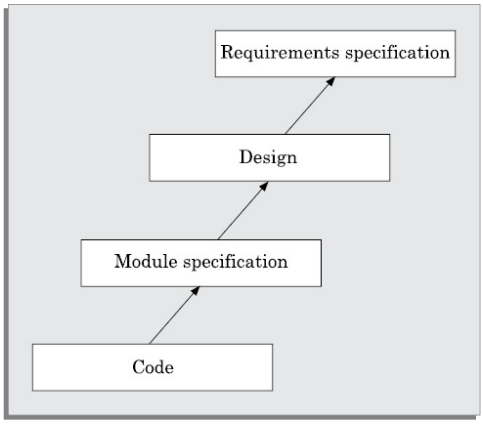


Figure13.1:Aprocess modelforreverseengineering.

- Afterthecosmeticchangeshavebeencarriedoutonalegacysoftware,theprocess of extracting the code, design, and the requirements specification can begin.
- TheseactivitiesareschematicallyshowninFigure 13.2.
- In order to extract the design, a full understanding of the code is needed. Some automatictoolscanbeusedtoderivethedataflowandcontrolflowdiagramfrom 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.

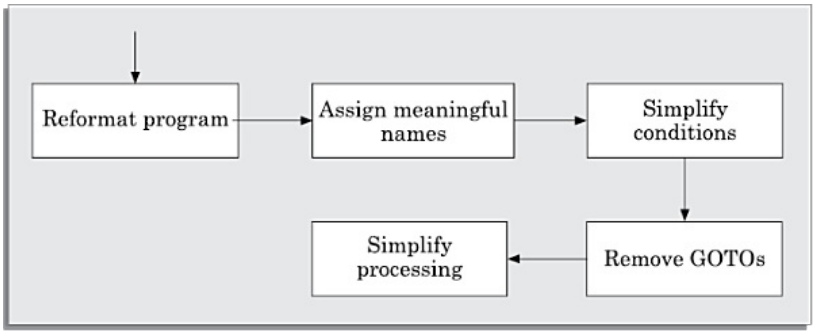


Figure13.2:Cosmeticchangescarriedoutbeforereverse engineering.

References

1. A Practitioners Approach-Software Engineering,- R.S. Pressman, McGraw Hill.
2. An Integrated Approach to Software Engineering – Pankaj Jalote, Narosa Publishing House, Delhi, 3rd Edition.
3. Software Engineering– K.K. Aggarwal and Yogesh Singh, New Age International Publishers, 3 rd edition.
4. Fundamentals of Software Engineering –Rajib Mall, PHI Publication,3rdEdition.

Prepared by

Dr.G.MuthuLakshmi B.E.,M.E.,Ph.D.
Associate Professor
Department of Computer Science & Engineering,
Manonmaniam Sundaranar University,
Abishekapatti, Tirunelveli - 627012,
Tamilnadu, India.