

# Software Engineering and Testing (SMCS51)

Software Engineering ..

By

Mrs. R.Waheetha, MCA, M.Phil  
Head,  
Department of Computer Science  
Holy Cross Home Science College  
Thoothukudi

# Unit I

## (syllabus)

- Introduction:- Evolution – From an Art form on Engineering Discipline: Evolution of an Art into an Engineering Discipline.
  - Software Development of Projects: Program versus Product –
  - Emergence of Software Engineering: Early Computer Programming – High Level Language Programming – Control Flow-based Design – Data Structure Oriented Design – Object Oriented Design. Software Life Cycle Models:- A few Basic Concepts – Waterfall Model and its Extension: Classical Waterfall Model – Iterative Waterfall Model – Prototyping Model – Evolutionary Model. – Rapid Application Development (RAD): Working of RAD. –Spiral Model.

# EVOLUTION—FROM AN ART FORM TO A N ENGINEERING DISCIPLINE

- Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals.
- Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.
- The early programmers used an *ad hoc programming style*.
- This style of program development is now variously being referred to as *exploratory, build and fix, and code and fix styles*.

- In a build and fix style, a program is quickly developed without making any specification, plan, or design.
- The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques solely guided by his own intuition, experience, whims, and fancies.
- The exploratory style comes naturally to all first time programmers.



# Evolution Pattern for Engineering Disciplines

- The evolution of the software development styles over the last sixty years, tells that it has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline.
- Every technology in the initial years starts as a form of art.
- Over time, it graduates to a craft and finally emerges as an engineering discipline.
- Those who knew iron making, kept it a closely-guarded secret.
- This esoteric knowledge got transferred from generation to generation as a family secret.

- Slowly, over time technology graduated from an art to a craft form where tradesmen shared their knowledge with their apprentices and the knowledge pool continued to grow.
- In the early days of programming, there were good programmers and bad programmers.
- The good programmers knew certain principles (or tricks) that helped them write good programs, which they did not share with the bad programmers.
- Over the next several years, all good principles were organised into a body of knowledge that forms the discipline of software engineering.

# A Solution to the Software Crisis

- Software engineering is one options that is available to tackle the present software crisis.
- The expenses that organizations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing an worrying trend over the years
- The trend of increasing software costs is probably the most vexing.
- Hardware Prices would become insignificant compared to software prices—when you buy any software product the hardware on which the software runs would come free with the software!!!

## Factors that contribute to the present software crisis are

- Rapidly increasing problem size
- Lack of adequate training in software engineering techniques
- Increasing skill
- Shortage and low productivity improvements.
- What is the remedy?
  - It is believed that a satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the developers, along with the further advancements .

# SOFTWARE DEVELOPMENT PROJECTS

- Programs *versus Products*

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

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

- A professional software is developed by a group of software developers working together in a team. I
- So we have to use some systematic development methodology.
- Else they would find it very difficult to interface and understand each other's work, and produce a coherent set of documents.
- However, when developing small programs for personal use, rigid adherence to software engineering principles is often not worthwhile.
- An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate.



# EMERGENCE OF SOFTWARE ENGINEERING

## 1. Early Computer Programming

- Early commercial computers were slow and elementary.
- It took a lot of time for computation.
- Programs were very small in size and were written in assembly language.
- Programmers wrote them without proper plan, design, etc.

## 2. High Level Language Programming

- Computers became faster with the introduction of this semiconductor technologies.
- This helped to solve more complex problems.
- At this time, high level language BASIC, FORTRAN, COBOL were introduced.

## 3. Control Flow-Based Design

- Programmers found it difficult to write cost effective and correct program.
- They also found it difficult to understand and maintain program written by others.
- So they started to pay attention to program's control flow structure.
- Thus flow charting technique was developed. Eg: fig 1.8.

## 4. Data Structure – Oriented Design

- While developing program, it was found that attention should be paid on data structure.
- An example of data structure oriented design is JSP ( Jackson's Structured Programming ).
- This helped to derive the program structure from its data structure representation.

## 5. Data Flow – Oriented Design

- In this, the major data items handled must be identified and then the processing required on these data items to produce required output must be determined.
- The functions (processes) and the data items that are exchanged between the different function are represented as Data Flow Diagram (DFD) .

## 6. Object – Oriented Design

- It is a technique which deals with natural objects. Problems are identified and then relationship among the objects like composition, reference and inheritance are determined.
- It has gained good acceptance because of its simplicity, scope for code, design reuse, lower development cost and easy

# **SOFTWARE LIFE CYCLE MODELS**

# Software life cycle

- All living organisms undergo a life cycle.
- example when a seed is planted, it germinates, grows into a full tree, and finally dies.
- The term *software life cycle* has been defined to imply the *different* over which a software evolves from an initial customer request , then fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.


- The life cycle of every software starts with a request for it by one or more customers.
- At this stage, the customers are not clear about all the features that are needed.
- They can only vaguely describe what is needed.
- This is the stage where the customer feels a need for the software and forms rough ideas about the required features is known as the *inception stage*.
- *In the inception stage, a software evolves through a series of identifiable stages (also called phases).*
- Then development activities are carried out by the developers, until it is fully developed and is released to the customers.
- Once installed, it is made available for use, the users start to use the software.
- This is the start of the operation (also called *maintenance* ) phase.



- As the users use the software, they request for fixing any failures that they find.
- They also continually suggest several improvements and modifications to the software.
- Thus, the maintenance phase involves continually making changes to the software to accommodate the bug-fix and change requests from the user.
- The operation phase is usually the longest of all phases and constitutes the useful life of a software.
- Finally the software is retired, when the users do not find it any longer useful due to reasons such as changed business scenario, availability of a new software having improved features and working, changed computing platforms, etc.
- This forms the essence of the life cycle of every software.

# Software development life cycle (SDLC) model

- A *software development life cycle (SDLC) model* describes the different activities that is needed to be carried out for the software to evolve .
- *Software development life cycle (SDLC) and software development process interchangeable* from a software development process.
- A software development process describes the life cycle activities more precisely and elaborately, as compared to an SDLC.
- A development process also prescribe a specific methodologies to carry out the activities, and also recommends the specific documents and other artifacts that should be produced at the end of each phase.
- The term SDLC can be considered to be a more generic term, as compared to the development process

- 
- 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

- 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, describes the steps to carry out only a single or at best a few individual activities.

# Why use a development process?

- The primary advantage of using a development process is that it encourages development of software in a systematic and disciplined manner.
- It is important that the development of professional software need team effort.
- When software is developed by a team than individual programmer, use of a life cycle model becomes successful completion of the project.
- Software development organizations have realized that suitable life cycle model helps to produce good quality software and that helps minimize the chances of time and cost overruns.

- Suppose a single programmer is developing a small program.
- For example, a student may be developing code for a class room assignment.
- The student might succeed even when he does not strictly follow a development process and adopts a build and fix style of development.
- What difficulties will 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 idea.
- A software development problem has been divided into several parts and these parts are assigned to the team members.

- Suppose the team members are given freedom to develop the parts assigned to them in whatever way they like.
- It is possible that one member might start writing the code for his part while making assumptions about the input results required from the other parts, another might decide to prepare the test documents first, and some other developer might start to carry out the design for the part assigned to him.
- In this case, severe problems can arise in interfacing the different parts and in managing the overall development. T
- Therefore, *ad hoc development turns out* to be is a sure way to have a failed project.
- 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.
- 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.
- 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 smaller 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?

- An organisation must have not only well-defined development process, but the development process needs to be properly documented.
- Consider development organisation which does not document its development process.
- In this case, its developers develop only 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 few important problems that may come across when a development process is not adequately documented. Some are:
  - 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.

- Eg : 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.
- Also, they would debate whether the test cases should be documented at all.
- 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 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 properly.

- 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, 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 doubt the capability of developing quality software and the organisation might find it difficult to win tenders for software development.
- Nowadays, good software development organisations document their development process in the form of a booklet.

# Phase entry and exit criteria

- A good SDLC should 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 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 there is scope for ambiguity in starting and ending various phases, and cause a lot of confusion among the developers.
- Sometimes they might stop the activities in a phase, and some other times may take more time than the phase should have been over.
- The decision regarding whether a phase is complete or not becomes difficult for the project manager to accurately tell how much has the development progressed.
- 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*.



# Classical Waterfall Model



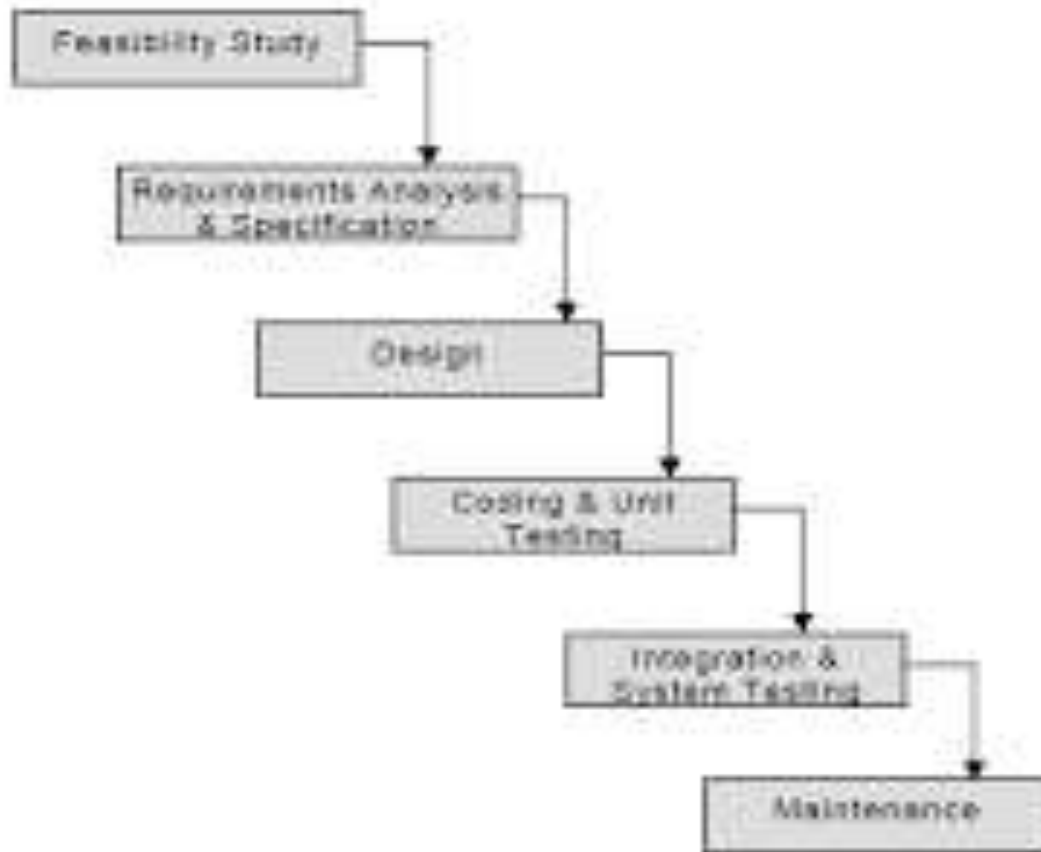


Fig 2.1: Classical Waterfall Model

The different phases of this model are

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

# Feasibility study

- It is to determine if it is financially and technically feasible to develop product.
- It involves analysis of problem and collection of relevant information. Collected data are analyzed to get.
- An abstract problem definition:
  - Only important requirements of customers are collected others are ignored.
  - Formulation of the different strategies for solving the problem.
  - Evaluation of different solution strategies. i.e. estimates of resource required, cost, time, etc.

# Requirement analysis and specification

- It has two phase
  - Requirement gathering and analysis
  - Requirement specification

- Requirement gathering and analysis
  - The goal of requirement gathering is to collect relevant information from the customer with a clear view
- Requirement specification
  - Both analysis and gathering activity are organized into Software Requirement Specification (SRS) document. The three important contents of this documents are
    - Functional requirement
    - Non function requirement
    - Goals of implementation
- The SRS serves as a contract between development team and the customer.

# Design

- Goal of design is to transform the requirements in SRS document into a structure suitable for implementation. There are two approaches.
  - Traditional design approach
  - Object – Oriented design approach



- Traditional design approach:

- It is based on data – flow oriented design approach.
- Structured analysis is carried out followed by structured design activity.
- Data Flow Diagram (DFD) are used to perform structured analysis.
- Structured design has two activities i.e. architectural design and detailed design



- Object Oriented design approach:
  - Various objects that occur in the problem domain and solution domain are identified.
  - The relationship between these objects are identified.
  - It is further refined to obtain detailed design.

# Coding and unit testing

- The purpose of this phase is to translate the software design into source code.
- Each component of design is implemented as a program module.
- After coding is completed, each module is unit tested.
- The main objective of unit testing is to determine the correct working of individual modules.

# Integration and System testing

- During this phase, the different modules are integrated.
- It is carried out incrementally over a no. of slips.
- After integrating all modules system testing is carried out.
- There are three types of system testing.
  - $\alpha$ -testing - testing performed by the development team
  - $\beta$ -testing – testing performed by a friendly set of customer.
  - Acceptance testing – performed by the customer after product delivery to find whether to accept or reject it.

# Maintenance

- Maintenance requires more effort. It is roughly in 40:60 ratio. There are three kinds of activities.
  - Corrective maintenance
    - It involves in correcting the errors found during product development phase.
  - Perfective maintenance
    - It involves in improving and enhancing the functionalities of the system.
  - Adaptive maintenance
    - It is required for porting the software to work in a new environment.

# 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 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 incorporates no mechanism for error correction.

- Programmers are humans and as the old adage says *to err is humane*.
- *The cause for errors can be many—oversight, wrong interpretations, use of incorrect solution scheme, communication gap, etc.*
- These defects usually get detected much later in the life cycle like in coding or testing.
- Once a defect is detected at a later time, the developers need to redo some of the work done during that phase.
- Therefore, it becomes impossible to strictly follow the classical waterfall model of software development.

## ● **Difficult to accommodate change requests:**

- This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project.
- The customers' requirements usually keep on changing with time.
- But, in this model it is difficult to accommodate the requirement change requests made by the customer after the requirements specification phase is complete.

## ● **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.
- For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete.
- In this case, the activities of the design and testing phases overlap.
- Consequently, it is safe to say the different phases need to overlap for cost and efficiency reasons.

# Iterative Waterfall Model

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

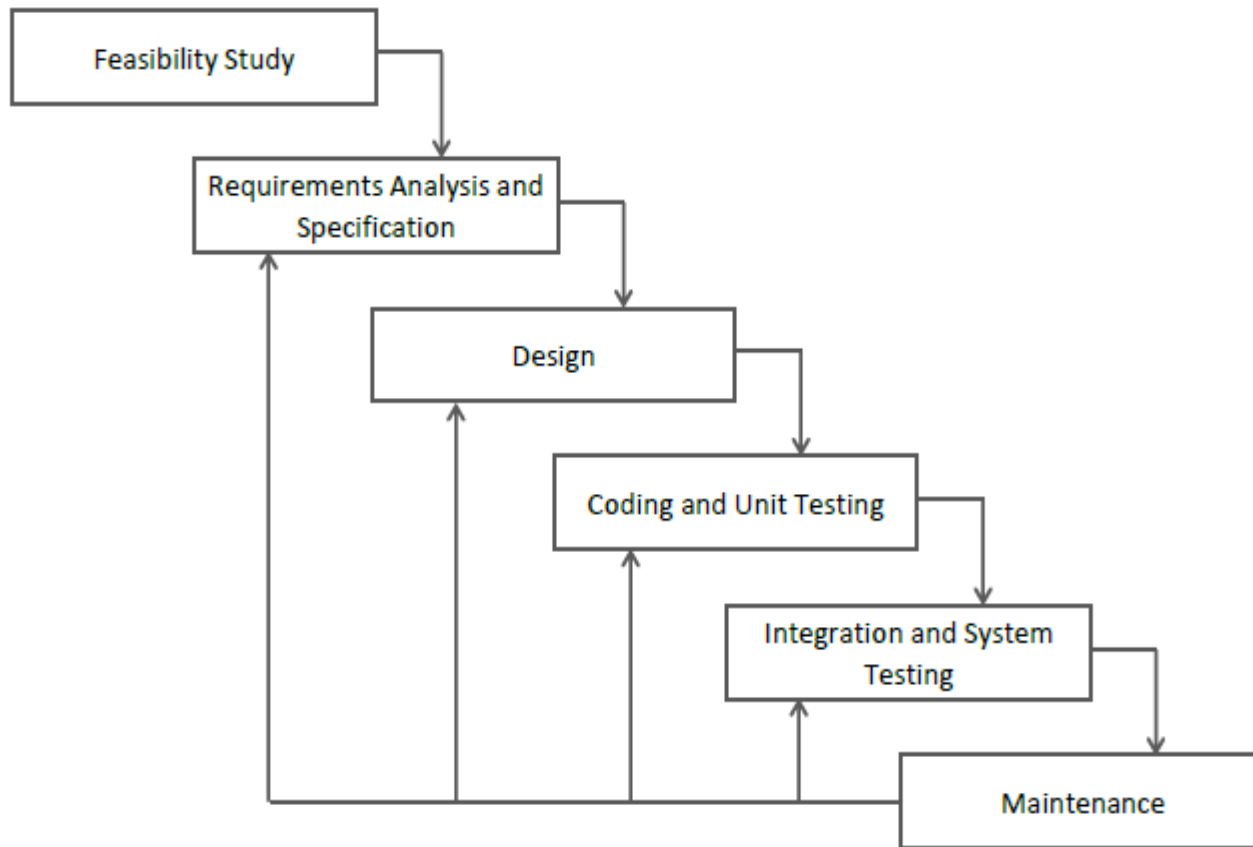


Figure 2: Iterative Waterfall Model

- The feedback paths allow for correction of the errors committed during a phase, as and when errors are detected in later phases i.e. it allows to correct the errors found in that phase.
- But there is no feedback path to the feasibility stage.
- **Phase Containment of Errors:**
  - Though errors cannot be avoided, it is desirable to detect the errors in the same phase in which they occur.
  - This can reduce the effort required for correcting bugs.
  - Eg. If a problem is found in design phase, it must be identified and corrected in that phase itself.
  - The errors should be detected as early as possible.
  - The principle of detecting errors as close to their point of introduction as possible is known as phase containment of errors.

## ● **How can phase containment of errors be achieved?**

- An important technique is frequently used to conduct review after every milestone.
- In spite of best effort to detect error in the same phase, still some errors can escape.
- So rework of already completed phase is required.
- Thus cannot complete phase at specified time.
- This makes the different life cycle phase overlap in time.

## • **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.
- The projects are now shorter, and involve Customised software development.
- Software was earlier developed from scratch.
- Now reuse of code is possible.
- The software services (customised software) are poised to become the dominant types of projects.

## ● **Difficult to accommodate change requests:**

- A major problem with the waterfall model is that the requirements need to be frozen before the development starts.
- Accommodating even small change requests after the development activities are difficult.
- Once requirements have been frozen, the waterfall model provides no scope for any modifications to the 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.



- **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.
- Strict adherence to the waterfall model creates blocking states.

## ● **Error correction unduly expensive:**

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

## ● **Limited customer interactions:**

- This model supports very limited customer interactions.
- It is generally accepted that software developed in isolation from the customer is the cause of many problems.
- 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 over emphasises 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.

# Prototyping Model

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

## • Necessity of the prototyping model

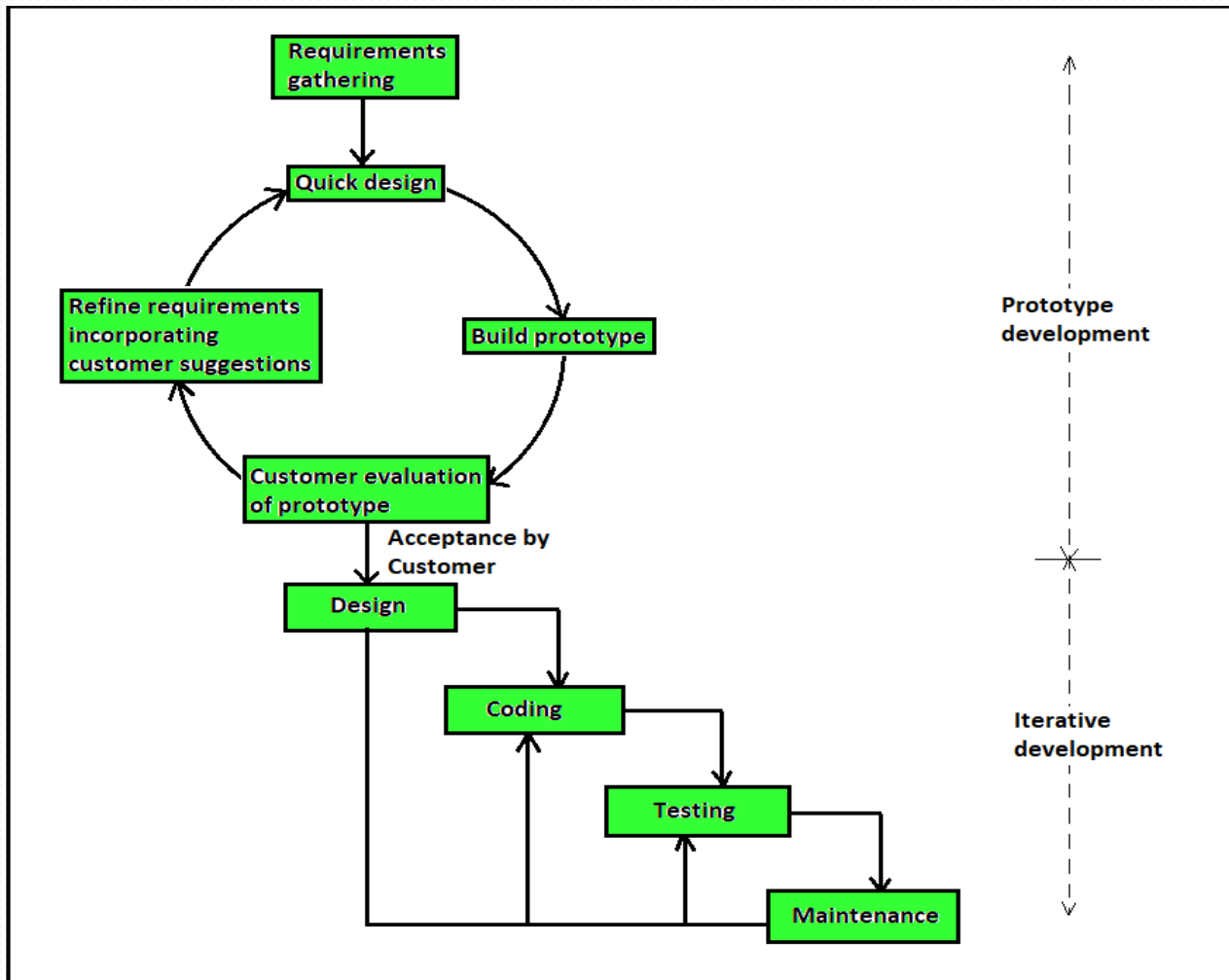
- We identify three types of projects for which the prototyping model can be followed to advantage:
- It is advantageous to use the prototyping model for development of the *graphical user interface (GUI) part of an application*.
- It is easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer.
- *It is much easier* to form an opinion regarding what would be more suitable by experimenting with a working user interface, rather than trying to imagine the working of a user interface.
- The prototyping model is especially useful when the exact technical solutions are unclear to the development team.
- A prototype can help them to critically examine the technical issues associated with product development.
- For example, consider a situation where the development team has to write a command language interpreter as part of a graphical user interface development.
- Suppose none of the team members has ever written a compiler before.

- This risk can be resolved by developing a prototype compiler for a very small language to understand the issues associated with writing a compiler for a command language.
  - Once they feel confident in writing compiler for the small language, they can use this knowledge to develop the compiler for the command language.
  - An important reason for developing a prototype is that it is impossible to “get it right” the first time.
  - 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
- 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



## ● **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.
- 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, 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.

## ● **Strengths of the prototyping model**

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

## ● **Weaknesses of the prototyping model**

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

# 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 that makes it difficult to accommodate any change requests from the customer.
- It has a few extensions from the waterfall model.
- It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions.
- In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer.
- But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction



The major goals of the RAD model are as follows:

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

# ● Main motivation


- In the iterative waterfall model, the customer requirements need to be gathered, analysed, documented, and signed off upfront, before any development could start.
- Often clients do not know what they exactly wanted until they saw a working system.
- It has now become practice that only through the process commenting on an installed application that the exact requirements can be brought out.
- Naturally, the delivered software often does not meet the customer expectations and many change request are generated by the customer.
- The changes are incorporated through subsequent maintenance efforts.
- This made the cost of accommodating the changes extremely high and it usually took a long time to have a good solution.
- The RAD model tries to overcome this problem by inviting and incorporating customer feedback on successively developed and refined prototypes.



# Working of RAD

- In the RAD model, development takes place in a series of short cycles or iterations.
- At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time.
- The time planned for each iteration is called a *time box*. *Each iteration is planned to enhance the* implemented functionality of the application by only a small amount.
- During each time box, a quick-and-dirty prototype-style software for some functionality is developed.
- The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary.



- 
- The prototype is refined based on the customer feedback.
  - 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:**
    - A customised software is developed for one or two customers only by adapting an existing software.
    - In customised software development projects, reuse is usually made of code from pre-existing software.
    - For example, a company might have developed a software for automating the data processing activities at one or more educational institutes.
    - When any other institute requests for an automation package to be developed, typically only a few aspects need to be tailored—since among different educational institutes, most of the data processing activities such as student registration, grading, fee collection, estate management, accounting, maintenance of staff service records etc. are similar to a large extent.
    - Projects involving such tailoring can be carried out speedily and cost effectively using the RAD model

- **Non-critical software:**

- The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery.
- The developed product is usually far from being optimal in performance and reliability.
- For well understood development projects and where the scope of reuse is rather restricted, the Iterative waterfall model may provide a better solution.

- **Highly constrained project schedule:**

- RAD aims to reduce development time at the expense of good documentation, performance, and reliability.
- 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):**
  - Software products are generic in nature and usually have wide distribution.
  - For such systems, optimal performance and reliability are imperative in a competitive market.
  - The RAD model of development may not yield systems having optimal performance and reliability.
- **Requirement of optimal performance and/or reliability:**
  - For certain categories of products, optimal performance or reliability is required.
  - Examples of such systems include an operating system (high reliability required) and a flight simulator software (high performance required).
  - If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

- **Lack of similar products:**

- If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts.
- In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.

- **Monolithic entity:**

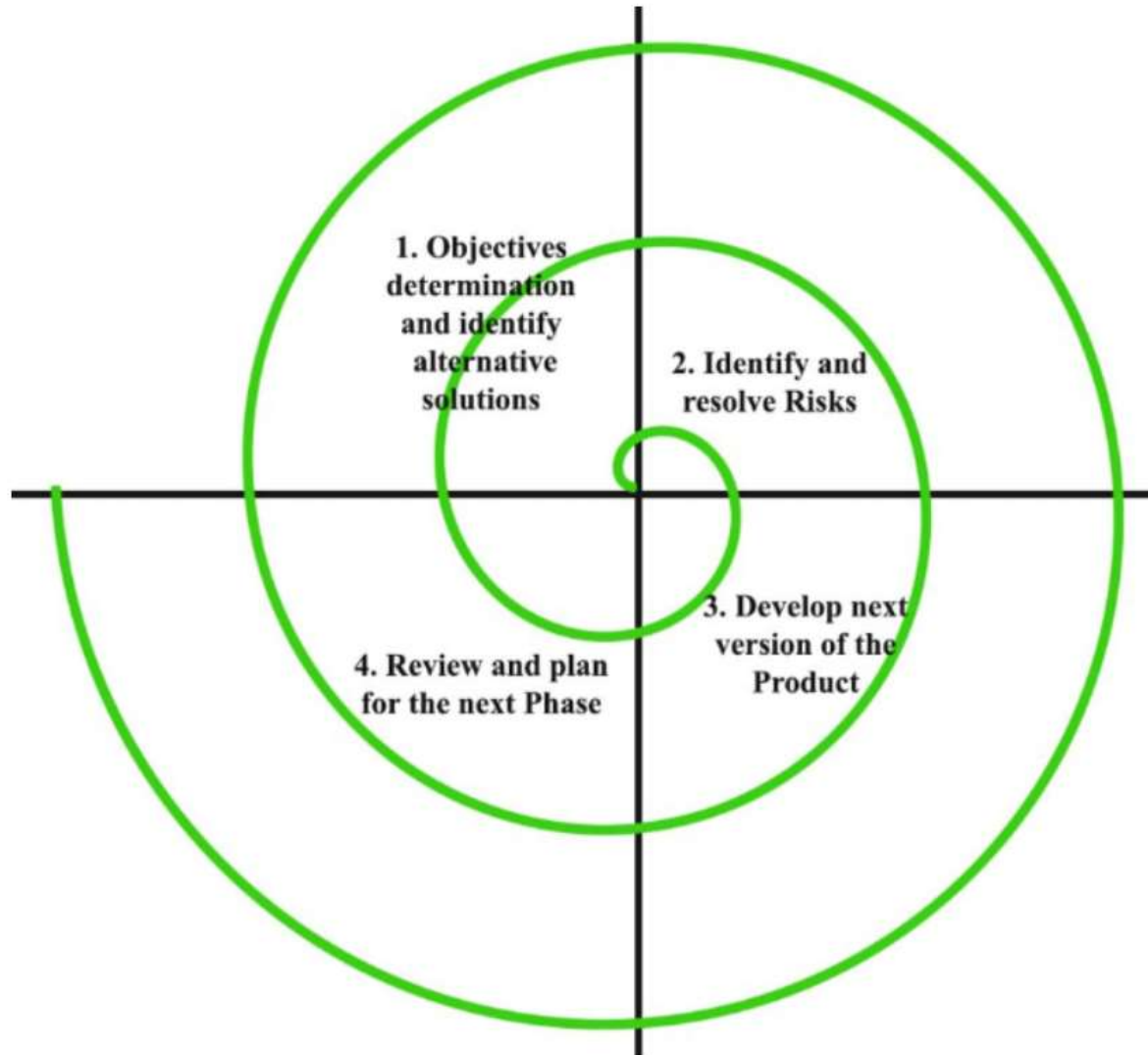
- For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered.
- In this case, it becomes difficult to develop a software incrementally.



# **SPIRAL MODEL**



- This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops.
- The exact number of loops of the spiral is not fixed and can vary from 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.
- 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 presentation.
- 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.



## ● 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.
- 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) .
- 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.
- Implementation of the identified features forms a phase.
- **Quadrant 1:**
  - The objectives are investigated, elaborated, and analysed.
  - Based on this, the risks involved in meeting the phase objectives are identified.
  - In this quadrant, alternative solutions possible for the phase under consideration are proposed.
- **Quadrant 2:**
  - During the second quadrant, the alternative solutions are evaluated to select the best possible solution.
  - To be able to do this, the solutions are evaluated by developing an appropriate prototype.

- **Quadrant 3:**

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

- **Quadrant 4:**

- Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.
- The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.
- In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses.
- In this model, the project manager plays the crucial role of tuning the model to specific projects.
- To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified.

## • **Advantages/pros and disadvantages/cons of the spiral model**

- 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.
- 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.
- It is much more powerful than the prototyping model.
- All these risks are resolved by building a prototype before the actual software development starts.



# UNIT II (SMCA51)

Software Engineering

By

Mrs. R.Waheetha, MCA, M.Phil

Head,

Department of Computer Science

Holy Cross Home Science College

Thoothukudi

# Unit II (Syllabus)

- Software Project Management:- Responsibilities of a Software Project Manager – Project Planning- Project Estimation Techniques-Risk Management. Requirements Analysis and Specification:- Requirements Gathering and Analysis – Software Requirements Specifications (SRS):Users of SRS Document – Characteristics of a Good SRS Document – Important Categories of Customer Requirements – Functional Requirements – How to Identify the Functional Requirements? – Organisation of the SRS Document.



# RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

- **Job Responsibilities for Managing Software Projects**
  - A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description.
  - 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.
- We can broadly classify these activities into two major
  - Project planning, and
  - Project monitoring and control.

- Project Planning:
  - It involves estimating several characteristics of the project, planning the project activities.
  - It is under taken immediately after the feasibility study.
- Project Monitoring and Control activities:
  - It is undertaken once the development activities start.  
The aim of monitoring & control activities is to ensure that the development proceeds as per the plan.

# Skills Necessary for Software Project Management:

- Theoretical knowledge of different project management technique is necessary.
- Knowledge in cost estimation, risk management, configuration management etc. are need.
- Need good communication skill and the ability to get work done.
- Tracking and controlling the progress of project, customer interaction, team building are acquired through experience.

# 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.
- Project planning requires care and attention since commitment to unrealistic time and resource estimates result in schedule slippage.
- Schedule delays can cause customer dissatisfaction.
- It can even cause project failure.
- project planning is undertaken by the project managers with utmost care and attention.
- However, for effective project planning, 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?
- The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

- **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.
- Observe that size estimation is the first activity that a project manager undertakes during project planning.

# PROJECT ESTIMATION TECHNIQUES



- Estimation of various project parameters is an important project planning activity.
- The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost.
- Accurate estimation of these parameters is important because it forms the basis for resource planning and scheduling.
- These can broadly be classified into three main categories:
  - Empirical estimation techniques
  - Heuristic techniques
  - Analytical estimation techniques

# Empirical Estimation Techniques

- Empirical estimation techniques are essentially based on making an educated guess of the project parameters.
- While using this technique, prior experience with development of similar products is helpful.
- Although empirical estimation techniques are based on common sense and subjective decisions, the different activities involved in estimation have been formalised to a large extent.

# Heuristic Techniques

- Heuristic techniques assume that the relationships that exist among the different project parameters can be modeled using suitable mathematical expressions.
- Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression.
- Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.
- Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size.

- A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:
- Estimated Parameter =  $c_1 \times e^{d_1}$
- In the above expression, e represents a characteristic of the software that has already been estimated (independent variable).
- Estimated Parameter is 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 =  $c_1 \times p_1 d_1 + c_2 \times p_2 d_2 + \dots$   
 where,  $p_1, p_2, \dots$  are the basic (independent) characteristics of the software already estimated, and  $c_1, c_2, d_1, d_2, \dots$  are constants.

- Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters.
- The independent parameters influence the dependent parameter to different extents.
- This is modelled by the different sets of constants  $c_1$ ,  $d_1$ ,  $c_2$ ,  $d_2$ , .... Values of these constants are usually determined from an analysis of historical data.
- The dependent parameter to be estimated could be effort, project duration, staff size, etc.,  $c_1$  and  $d_1$  are constants.
- The values of the constants  $c_1$  and  $d_1$  are usually determined using data collected from past projects (historical data).

# EMPIRICAL ESTIMATION TECHNIQUES

- Empirical estimation techniques have, been formalised to a certain extent.
- Yet, these are still essentially euphemisms for pure guess work.
- These techniques are easy to use and give reasonably accurate estimates.
- Two popular empirical estimation techniques are—
  - Expert judgement
  - Delphi estimation techniques.

# RISK MANAGEMENT

- Every project is susceptible to a large number of risks.
- Without effective management of the risks, even the most meticulously planned project may go hay ware.
- **A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway.**
- If a risk becomes real, the anticipated problem becomes a reality and is no more a risk.
- If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project.
- Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that plans can be prepared beforehand to take care of risk.
- Risk management consists of three essential activities
  - risk identification,
  - risk assessment, and
  - risk mitigation.



# Risk Identification

- The project manager needs to anticipate the risks in a project as early as possible.
- As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks is minimised.
- Early risk identification is important.
- Risks that are likely to affect a project must be identified and listed.
- In order to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes.
- The project manager can then examine which risks from each class are relevant to the project.
- There are three main categories of risks which can affect a software project:
  - project risks,
  - technical risks, and
  - business risks.

# ● Project risks:

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



- **Technical risks:**

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

- **Business risks:**

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

# Classification of risks in a project

- **Example** Let us consider a satellite based mobile communication Product
- The project manager can identify several risks in this project.
- What if the project cost moves up and overshoots what was estimated?:
- **Project risk.**
- What if the mobile phones that are developed become too bulky in size to conveniently carry?:
- **Business risk.**
- What if it is later found out that the level of radiation coming from the phones is harmful to human being?:
- **Business risk.**
- What if call hand-off between satellites becomes too difficult to implement?:
- **Technical risk.**
- In order to be able to successfully foresee and identify different risks that might affect a software project, it is a good idea to have a company disaster list.
- This list would contain all the bad events that have happened to software projects of the company over the years including events that can be laid at the customer's doors.
- This list can be read by the project managers in order to be aware of some of the risks that a project might be susceptible to. Such a disaster list has been found to help in performing better risk analysis.

# 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.

# 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.

## ● **Avoid the risk:**

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

# Risk reduction

- 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.
- 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.





# **REQUIREMENTS ANALYSIS AND SPECIFICATION**

- Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, thereby pushes up the development costs.
- This also sets the customer dissatisfaction and bitter customer-developer disputes and legal battles.
- Experienced developers consider the requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.
- Experienced developers take considerable time to understand the exact requirements of the customer and to document those.
- They know that without a clear understanding of the problem and proper documentation , it is impossible to develop a satisfactory solution.

- For any type of software development project, availability of a good quality requirements document is a key factor in the successful completion of the project.
- It also serves as the basis for various activities carried out during later life cycle phases.
- When software is developed in a contract mode for some other organisation (that is, an outsourced project), the crucial role played by documentation of the precise requirements cannot be overstated.
- Even when an organisation develops a generic software product, the situation is not very different since some personnel from the organisation's own

## • **An overview of requirements analysis and specification phase**

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

- **Who carries out requirements analysis and specification?**
- Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site.
- The engineers who gather 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 remove inconsistencies, anomalies and incompleteness.
- They then proceed to write the *software requirements specification (SRS) document*.

## ● **How is the SRS document validated?**

- It is first reviewed internally by the project team to ensure accuracy.
- It checks if it has captured all the user requirements.
- It also checks if 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.
- It also serves as a contract document between the customer and the development organisation

- **What are the main activities carried out during requirements analysis and specification phase?**
- Requirements analysis and specification phase mainly involves carrying out the following two important activities:
  - Requirements gathering and analysis
  - Requirements specification

# ● REQUIREMENTS GATHERING AND ANALYSIS

- The complete set of requirements are not available in the form of a single document from the customer.
- The complete requirements are rarely obtainable from any single customer representative.
- the requirements have to be gathered by the analyst from several sources in bits and pieces.
- These gathered requirements need to be analysed to remove several types of problems.
- We can divide the requirements gathering and analysis activity into two separate tasks:
  - *Requirements gathering*
  - *Requirements analysis*



# Requirements Gathering

- Requirements gathering is also popularly known as *requirements elicitation*.
- The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.
- A stakeholder is a source of the requirements and is a person, or a group of persons who either directly or indirectly are concerned with the software.
- It is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents.
- Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

- If customer wants to automate some activity in his organisation that is currently being carried out manually, then a working model of the system( manual system) exists.
- This is of great help in requirements gathering.
- Consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office.
- The analyst have to study the input and output forms and then understand how the outputs are produced from the input data.
- If a project involves in developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult.
- In the absence of a working system, more imagination and creativity is required on the part of the system analyst.

- Before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all information.
- During visit to the customer site, the analysts normally interview the end-users and customer representatives.
- Requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis are done.
- Many customers describe their requirements very vaguely.
- Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities and make the functionalities more general and more complete.

# **Requirement Gathering**

# 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.
- These documents may 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:

- There are many different categories of users of a software.
- Each category of users requires a different set of features from the software.
- It is important for the analyst to first identify the different categories of users and then determine the requirements of each.
- For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants.
- The library members would like to use the software to query availability of books and issue and return books.
- The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc.
- The 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*.
- The software performs various tasks of the users.
- The analyst tries to identify and understand the different tasks to be performed by the software.
- For each identified task, the analyst formulate the different steps necessary to realise the required functionality in consultation with the users.
- Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.



## ● **Scenario analysis:**

- A task can have many scenarios of operation.
- For different types of scenarios of a task, the behaviour of the software can be different.
- For example, the possible scenarios for the book issue task of a library automation software may be:
- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.
- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.
- For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users.
- For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.



## ● **Form analysis:**

- Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.
- During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications.
- In form analysis the existing 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.

# Requirements Analysis

- Requirements to form a clear understanding of the exact customer requirements and to remove any problems in the gathered requirements.
- The data collected from various stakeholders contain several contradictions, ambiguities, and incompleteness, since each stakeholder has only a partial and incomplete view of the software.
- It is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.
- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

- For carrying out requirements analysis effectively, the analyst first needs to develop a clear grasp of the problem.
- The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:
- What is the problem?
- Why is it important to solve the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the possible procedures that need to be followed to solve the problem?
- What are the likely complexities that might arise while solving the problem?
- 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.
- The following is an examples of anomalous requirements:
- **Example**
- While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder:
- When the temperature becomes high, the heater should be switched off.
- Please note that words such as “high”, “low”, “good”, “bad” etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted.
- If the threshold above which the temperature can be considered to be high is not specified, then it can be interpreted differently by different developers.

- **Inconsistency:**
- Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
- The following is an example of inconsistent requirements:
- **Example 4.3**
- Consider the following two requirements that were collected from two different stakeholders in a process control application development project.
- The furnace should be switched-off when the temperature of the furnace rises above 500 C.
- When the temperature of the furnace rises above 500 C, the water shower should be switched-on and the furnace should remain on.



- **Incompleteness:**

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

- The following is an example of incomplete requirements:
- **Example**
- Suppose one of the clerks expressed the following—
- If a student secures a *grade point average (GPA) of less than 6, then* the parents of the student must be intimated about the performance through a (postal) letter as well as through e-mail.
- However, on an examination of all requirements, it was found that there is no provision by which either the postal or e-mail address of the parents of the students can be entered into the system.
- The feature that would allow entering the e-mail ids and postal addresses of the parents of the students was missing, thereby making the requirements incomplete.



# SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

- After the analyst gather all the required information regarding the software 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.
- SRS document is the most important document and is the toughest to write.
- One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience.

# Users of SRS Document

- **Users, customers, and marketing personnel:**
  - These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs.
  - The customer may not be the user of the software, but may be some one employed or designated by the user.
  - The marketing personnel need to understand the requirements that they can explain to the customers.
- **Software developers:**
  - The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- **Test engineers:**
  - The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working.
  - They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
- **User documentation writers:**
  - The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- **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 understand the functionalities supported by the system.
- A clear knowledge of the functionalities can help them to understand the design and code.
- Many software engineers in a project consider the SRS document to be a reference document.
- However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer.
- The SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future.
- The SRS document can also be used as a legal document to settle disputes between the customers and the developers in a court of law.
- Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the requirements mentioned in the SRS document.

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

- **Forms an agreement between the customers and the developers:**
  - A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.
- **Reduces future reworks:**
  - The process of preparation of the SRS document forces the stakeholders to think about all of the requirements before design and development .
  - This reduces later redesign, recoding, and retesting.
  - Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

- **Provides a basis for estimating costs and schedules:**
  - Project managers usually estimate the size of the software from an analysis of the SRS document.
  - Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development.
  - The SRS document also serves as a basis for price negotiations with the customer.
  - The project manager also uses the SRS document for work scheduling.
- **Provides a baseline for validation and verification:**
  - The SRS document provides a baseline against which compliance of the developed software can be checked.
  - It is also used by the test engineers to create the *test plan*.
- **Facilitates future extensions:**
  - The SRS document usually serves as a basis for planning future enhancements.
  - Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

# 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.
- The analyst should be aware of the desirable qualities that every good SRS document should possess.
- IEEE Recommended Practice for Software Requirements Specifications describes the content and qualities of a good software requirements specification (SRS).
- Some of the identified desirable qualities of an SRS document are the following:



- **Concise:**

- The SRS document should be concise and at the same time unambiguous, consistent, and complete.
- 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.
- That is, 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.
- The SRS document is also called the *black-box specification of the software being* design elements that implement it.
- *Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases.*
- *Traceability is also important to verify the results of a phase with respect to the previous phase and to analyse the impact of changing a requirement on the design elements and the code.*



- **Modifiable:**

- Customers frequently change the requirements during the software development due to a variety of reasons.
- Therefore, 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.
- The SRS document should be easily modifiable.
- An SRS document should be well-structured.
- Having the description of a requirement scattered across many places in the SRS document may not be wrong—but it tends to make the requirement difficult to understand and also any modification to the requirement would become difficult as it would require changes to be made at large number of places in the document.

- **Identification of response to undesired events:**

- The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

- **Verifiable:**

- All requirements of the system as documented in the SRS document should be verifiable.
- 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.

# Attributes of Bad SRS Documents

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

## ● **Over-specification:**

- It occurs when the analyst tries to address the “how to” aspects in the SRS document.
- Over-specification restricts the freedom of the designers in arriving at a good design solution.

## ● **Forward references:**

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

## ● **Wishful thinking:**

- This type of problems concern description of aspects which would be difficult to implement.

## ● **Noise:**

- The term noise refers to presence of material not directly relevant to the software development process.
- Several other “sins” of SRS documents can be listed and used to guard against writing a bad SRS document and is also used as a checklist to review an SRS document.

# Important Categories of Customer Requirements

- A good SRS document, should properly categorize and organise the requirements into different sections.
- 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.
- It is useful to consider a software as offering a set of functions  $\{f_i\}$  to the user.
- These functions can be considered similar to a mathematical function  $f : I \rightarrow O$ , meaning that a function transforms an element  $(i_i)$  in the input domain (I) to a value  $(o_i)$  in the output (O).
- *This functional* view of a system is shown schematically in Figure 4.1.
- Each function  $f_i$  of the system can be considered as reading certain data  $i_i$ , and then transforming a set of input data  $(i_i)$  to the corresponding set of output data  $(o_i)$ .
- *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.

## • **Design and implementation constraints:**

- Design and implementation constraints are an important category of non-functional requirements.
- 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.



## ● **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 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.

# ● Goals of implementation

- The ‘goals of implementation’ which is part of the SRS document offers some general suggestions regarding the software to be developed.
- For example, the developers may use these suggestions while choosing among different design solutions.
- 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 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*.

# Functional Requirements

- In order to document the functional requirements of a system, it is necessary to identify the high-level functions of the systems.
- The high-level functions would be split into smaller sub requirements.
- A high-level function is one using which the user can get some useful piece of work done.
- For example, how useful must a piece of work be performed by the system for it to be called ‘a useful piece of work’ ?
- Can the printing of the statements of the ATM transaction during withdrawal of money from an ATM be called a useful piece of work?
- Printing of ATM transaction should not be considered a high-level requirement, because the user does not specifically request for this activity.
- The receipt gets printed automatically as part of the withdraw money function.
- Usually, the user requests the services of each high-level requirement.

- To treat print expressed as transformation of some input data to some output data should be documented as the functional requirement.
- Any other requirements whose compliance by the developed system can be verified by inspecting the system are documented as non-functional requirements.
- The difference between non-functional requirements and guidelines is the following.
- Non-functional requirements would be tested for compliance, before the developed product is accepted by the customer whereas guideline, are customer request that are desirable to be done, but would not be tested during product acceptance.
- Functional requirements form the basis for most design and test methodologies.
- Therefore, unless the functional requirements are properly identified and documented, the design and testing activities cannot be carried out satisfactorily.

# 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.



# 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 the understanding of the problem.
- Each high-level requirement is a way of system usage by some user to perform some meaningful piece of work.
- It is often useful to first identify the different types of users who use the system and then try to identify the different services expected from the software by different types of users.



# How to Document the Functional Requirements?

- Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented.
- A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data.

# Organisation of the SRS Document

- Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst.
- Organisation of the SRS document depends on the preferences of the system analyst himself, and he is often guided by the policies and standards being followed by the development company.
- The three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.
- 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.
-

# ● Introduction

- **Purpose:**

- This section should describe where the software would be deployed and how the software would be used.

- **Project scope:**

- This section should briefly describe the overall context within which the software is being developed.

- **Environmental characteristics:**

- This section should briefly outline the environment (hardware and other software) with which the software will interact.

- **Overall description of organisation 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 systems, or it is a new software.

- **Product features:**

- This section should summarize the major ways in which the software would be used

- **User classes:**

- Various user classes that are expected to use this software are identified and described here.
- 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.

## • **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.

## ● **Document**

- This section should describe the non-functional requirements other than the design and implementation constraints and the external interface requirements.
- **Performance requirements:**
  - Aspects such as number of transaction to be completed per second should be specified here.
- **Safety requirements:**
  - Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here.
  - 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.



# **UNIT III**

# **SOFTWARE DESIGN(SMCA 51)**

Software Engineering

By

Mrs. R.Waheetha, MCA, M.Phil

Head,

Department of Computer Science

Holy Cross Home Science College

Thoothukudi

# Unit III (Syllabus)

- Software Design:- Overview of the Design Process: Outcome of the Design Process – Classification of Design Activities. – How to Characterize a good Software Design? Function-Oriented Software Design:- Overview of SA/SD Methodology – Structured Analysis – Developing the DFD Model of a System: Context Diagram – Structured Design – Detailed Design.

# OVERVIEW OF THE DESIGN PROCESS

- The design process essentially transforms the SRS document into a design document.

# Outcome of the Design Process

- **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.
- **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.

- 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.
  - 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.

# Classification of Design Activities

- A good software design is realised by using a single step procedure, it requires iterating over a series of steps called the design activities.
- Depending on the order in which various design activities are performed, it is classified into two important stages.
  - Preliminary (or high-level) design, and
  - Detailed design.
- 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.
- 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.
- 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.

# 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.
- It is not usually true that a criterion that is crucial for some application, needs to be almost completely ignored for another application.
- It is therefore clear that the criteria used to judge a design solution can vary widely across different types of applications.
- There is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application.
- Most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess.



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

# **UNIT IV**

# **USER INTERFACE DESIGN**

# **(SMCS51)**

Software Engineering

By

Mrs. R.Waheetha, MCA, M.Phil  
Head,  
Department of Computer Science  
Holy Cross Home Science College  
Thoothukudi

# Unit IV(Syllabus)

- User Interface Design:- Characteristics of a good User Interface - Basic Concepts – Types of User Interfaces – Fundamentals of Components based GUI Development: Window System. Coding and Testing:- Coding – Software Documentation – Testing: Basic Concepts and Terminologies – Testing Activities. – Unit Testing – Black-box Testing: Equivalence Class Partitioning – Boundary Value Analysis. – White-box Testing.

- The user interface portion of a software product is responsible for all interactions with the user.
- Almost every software product has a user.
- In the early days of computer, no software product had any user interface.
- Now, things are very different—almost every software product is highly interactive.
- The user interface part of a software product is responsible for all interactions with the end-user.
- No wonder then that many users often judge a software product based on its user interface.
- Aesthetics apart, an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction.
- Normally, when a user starts using a system, he builds a mental model of the system and expects the system behaviour to conform to it.
- Sufficient care and attention should be paid to the design of the user interface of any software product.

- Systematic development of the user interface is also important from another consideration.
- Development of a good user interface usually takes significant portion of the total system development effort.
- For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part.
- Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously.
- Therefore, it is necessary to carefully study various concepts associated with user interface design and understand various systematic techniques available for the development of user interface.
- In this chapter, we first discuss some common terminologies and concepts associated with development of user interfaces.
- Then, we classify the different types of interfaces commonly being used. We also provide some guidelines for designing good interfaces, and discuss some tools for development of graphical user interfaces (GUIs). Finally, we present a GUI development methodology.

# CHARACTERISTICS OF A GOOD USER INTERFACE

- Before knowing about how to develop user interfaces, it is important to identify the different characteristics that are usually desired of a good user interface.
- Unless we know what exactly is expected of a good user interface, we cannot possibly design one.
- we identify a few important characteristics of a good user interface:

## • **Speed of learning:**

- A good user interface should be easy to learn.
- Speed of learning is hampered by complex syntax and semantics of the command issue procedures.
- A good user interface should not require its users to memorise commands.
- Besides, the following three issues are crucial to enhance the speed of learning:
  - Use of metaphors<sup>1</sup> and intuitive command names
  - Consistency
  - Component-based interface



## • **Use of metaphors<sup>1</sup> and intuitive command names:**

- Speed of learning an interface is greatly facilitated if these are based on some day to-day real-life.
- 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.
- 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.



- **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.
- 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.
- 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.

- **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.

- **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.

# 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”.
- A good online 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.
- 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.
- For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or interface

## ● **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 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.

## ● **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.

# 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 one of the biggest advantages of GUI since the user has the flexibility to simultaneously interact with several related items at any time .
- 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 very easy and the user can 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 bitmapped 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.

# TYPES OF USER INTERFACES

- User interfaces can be classified into the following three categories:
  - Command language-based interfaces
  - Menu-based interfaces
  - Direct manipulation interfaces
- Most modern applications use a careful combination of all these three types of user interfaces for implementing the user command .
- A study of the basic characteristics and the relative advantages of different types of interfaces would give a fair idea to the designer regarding which commands should be supported using what type of interface.

# Command Language-based Interface

- A command language-based interface is based on designing a command language which the user can use to issue the commands.
- A simple command language-based interface might simply assign unique names to the different commands.
- 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 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.



- The command language interface allows for most efficient command issue procedure requiring minimal typing.
- A command language-based interface can be implemented even on cheap alphanumeric terminals.
- A command language-based interface is easier to develop compared to a menu-based.
- However, command language-based interfaces suffer from several drawbacks.
- Command language-based interfaces are difficult to learn and require the user to memorise the set of primitive commands.
- Most users make errors while formulating commands in the command language and also while typing them.
- 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.
- For inexperienced users, command language-based interfaces are not suitable.



## • **Issues in designing a command language-based interface**

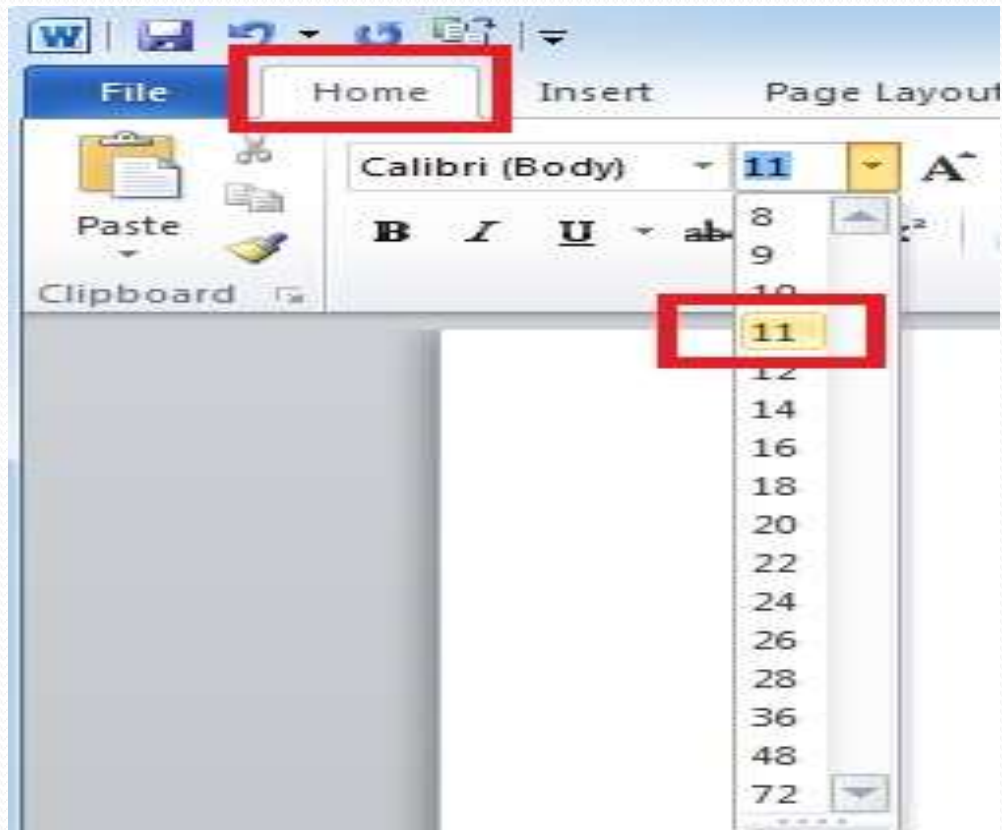
- Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimise the total typing required.
- We elaborate these considerations in the following:
- The designer has to decide what mnemonics (command names) to use for the different commands.
- 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

- In a menu-based interface the users does not require to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- In a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device.
- Experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast.
- 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 difficult to specify in a menu based system.
- A major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.

- 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.
  - 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.
  - 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.

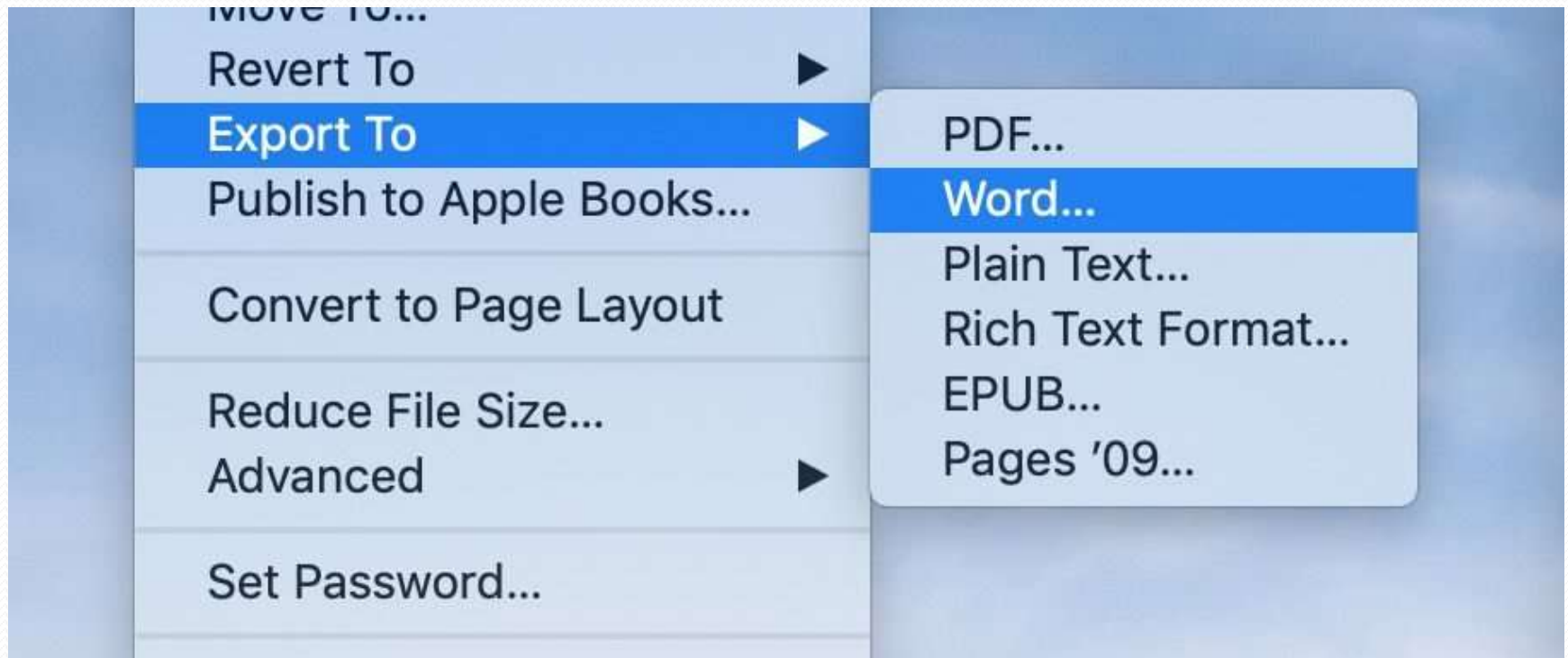
- **Font size selection using scrolling menu.**



## • **Walking menu:**

- **Walking menu is very commonly used to structure a large** collection of menu items.
- In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.
- An example of a walking menu is shown in Figure 9.2
- A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after all limited.

- **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 submenus 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.



## ● **Direct Manipulation Interfaces**

- Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons or objects).
- 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 is that the icons can be recognised by the users very easily, and that icons are language independent.
- It is difficult to give complex commands using a direct manipulation interface.





# **FUNDAMENTALS OF COMPONENT- BASED GUI DEVELOPMENT**

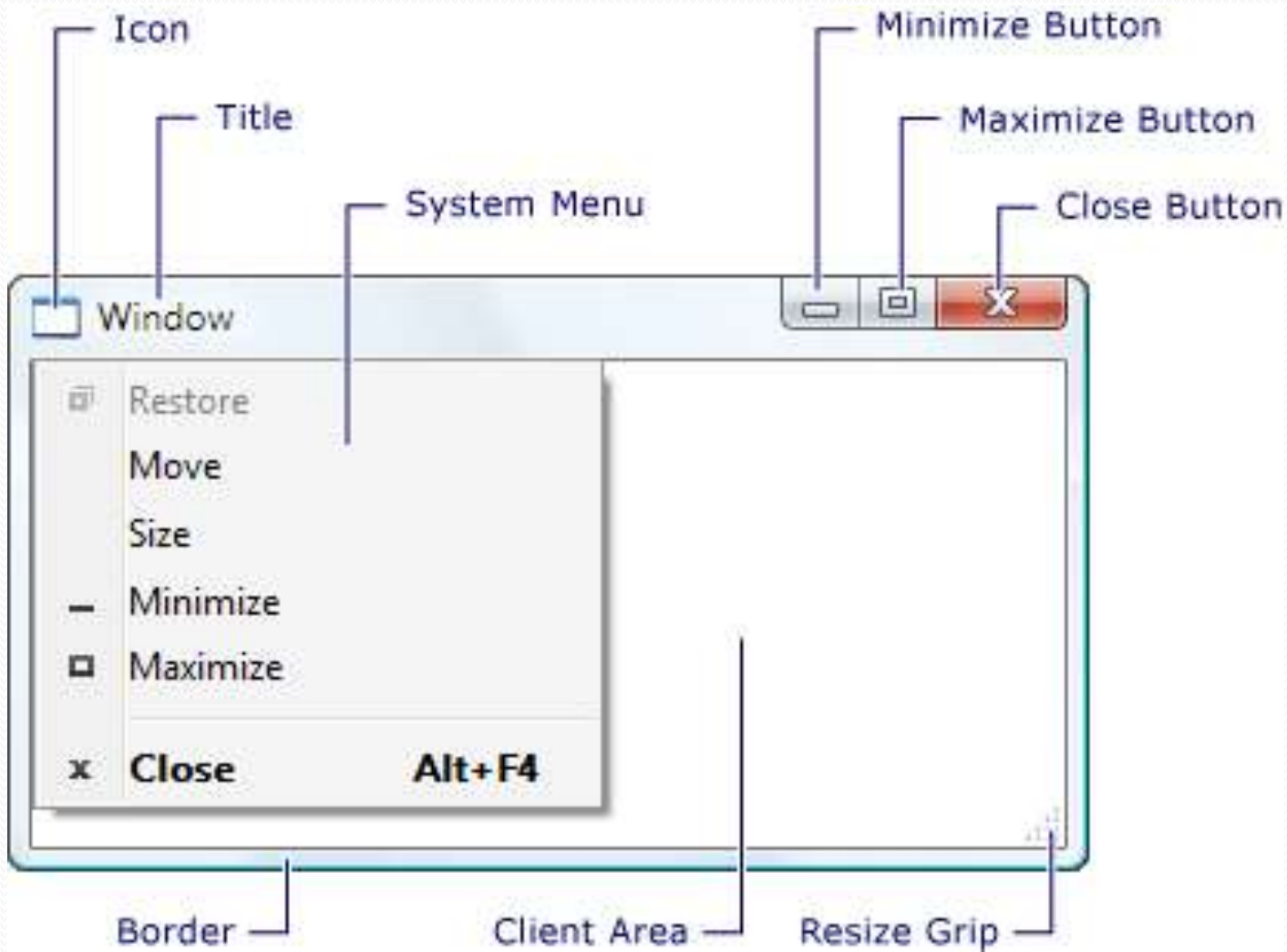
- Graphical user interfaces became popular in the 1980s.
- There were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive.
- The graphics terminals were of storage tube type and lacked raster capability.
- One of the first computers to support GUI-based applications was the Apple Macintosh computer.
- In early days of GUI design, the user interface programmer started his interface development from the scratch.
- They started from simple pixel display routines to write programs to draw lines, circles, text, etc.
- Then they developed their own routines to display menu items, make menu choices, etc.
- The current style of user interface development is component-based.
- Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system.
- The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

## ● Window System

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

## ● Window:

- **A window is a rectangular area on the screen. A window can be** considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.
- A window can be divided into two parts—client part, and non-client part.
- The client area makes up the whole of the window, except for the borders and scroll bars.
- The client area is the area available to a client application for display.
- The non-client-part of the window determines the look and feel of the window. The look and feel defines a basic behaviour for all windows, such as creating, moving, resizing, iconifying of the windows.
- The window manager is responsible for managing and maintaining the non-client area of a window.
- A basic window with its different parts is shown in Figure 9.3.



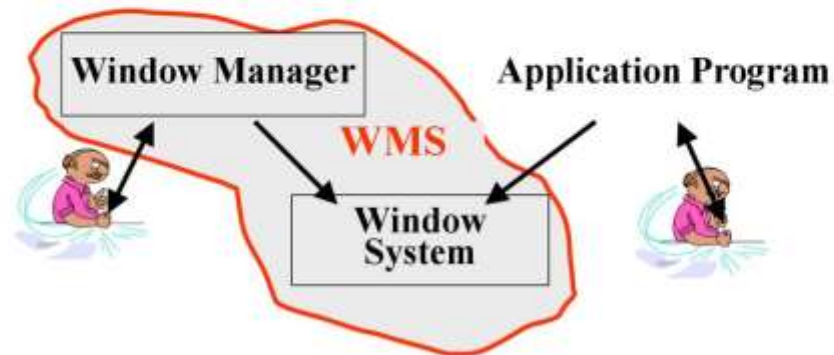
# ● Window management system (WMS)

- A graphical user interface typically consists of a large number of windows.
- 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.
- A WMS can be considered as a user interface management system (UIMS) —which also provides the basic behaviour to the windows and provides several utility routines to the application programmer for user interface development.
- A WMS simplifies the task of a GUI designer to a great extent by providing various windows such as move, resize, iconify, etc. 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

## Window management system (WMS)

- A WMS consists of two part:
  - a window manager
  - a window system.



- a window manager, and
- a window system.

- **Window manager and window system:**

- The window manager is built on the top of the window system that it makes use of various services provided by the window system.
- Several kinds of window managers can be developed based on the same window system.
- The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system.
- The application programmer can also directly invoke the services of the window system to develop the user interface.
- The relationship between the window manager, window system, and the application program is shown in Figure above.
- 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.



- 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.
- An object is 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.



## ● **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.
- The user can extend his knowledge of the behaviour of the standard components from one application to the other.
- The component-based user interface development style reduces the application programmer's work significantly.

# ● Visual programming

- Visual programming is the drag and drop style of program development.
- In this style , 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.

# CODING AND TESTING

- Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- In the coding phase, every module specified in the design document is coded and unit tested.
- During unit testing, each module is tested in isolation from other modules.
- After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.
- The integration plan, according to which different modules are integrated together, usually integration of modules goes through number of steps.
- The full product takes shape only after all the modules have been integrated together.
- System testing is conducted on the full product.
- During system testing, the product is tested against its requirements as recorded in the SRS document.

# CODING

- The input to the coding phase is the design document produced at the end of the design phase.
- The design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design.
- The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified.
- During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
- The objective of the coding phase is to transform the design of a system into code in a high-level language,
- These software development organisations formulate their own coding standards that suit them the most, and require their developers to follow the standards.

- The main advantages of 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.
- 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.
- After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing.

# Coding Standards and Guidelines

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



# ● **Representative coding standards**

## ● **Rules for limiting the use of globals:**

- These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.

## ● **Standard headers for different modules:**

- The header of different modules should have standard format and information for ease of understanding and maintenance.
- The following is an example of header format that is being used in some companies:
  - Name of the module.
  - Date on which the module was created.
  - Author's name.
  - Modification history.
  - Synopsis of the module. This is a small writeup about what the module does.
  - Different functions supported in the module, along with their input/output parameters.
  - Global variables accessed/modified by the module.

- **Naming conventions for global variables, local variables, and constant identifiers:**
  - A popular naming convention is that variables are named using mixed case lettering.
  - Global variable names would always start with a capital letter (e.g., GlobalData) and local variable names start with small letters (e.g., localData).
  - Constant names should be formed using capital letters only (e.g., CONSTDATA).
- **Conventions regarding error return values and exception handling mechanisms:**
  - The way error conditions are reported by different functions in a program should be standard within an organisation.
  - For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code.
  - This facilitates reuse and debugging.

## • **Representative coding guidelines:**

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

## • **Do not use a coding style that is too clever or too difficult to understand:**

- Code should be easy to understand.
- Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code.
- Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.

- **Avoid obscure side effects:**

- The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations.
- 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.
- It would be really hard to understand the code.

- **Do not use an identifier for multiple purposes:**

- Programmers often use the same identifier to denote several temporary entities.
- For example, some programmers make use of a temporary loop variable for also computing and storing the final result.
- The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used.
- However, there are several things wrong with this approach and hence should be avoided.

- Some of the problems caused by the use of a variable for multiple purposes are as follows:
- Each variable should be given a descriptive name indicating its purpose.
- This is not possible if an identifier is used for multiple purposes.
- Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult.

- **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 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.

# TESTING



- The aim of program testing is to help realise identify all defects in a program.
- However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free.
- This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume.
- Consider a function taking a floating point number as argument. If a tester takes 1sec to type in a value, then even a million testers would not be able to exhaustively test it after trying for a million number of years.
- Even with this obvious limitation of the testing process, we should not underestimate the importance of testing.
- We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.



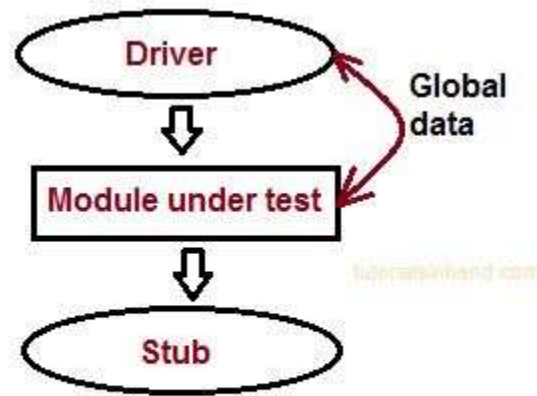
# Testing Activities

- 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 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.

# 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.
- **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.
  - 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 are usually not available until they too have been unit tested.
    - Stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

- Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.
- **Stub:**
  - The role of stub and driver modules is pictorially shown in Figure .



**Unit testing using Driver & 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.



- **Driver:**

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

# BLACK-BOX TESTING

- In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required.
- The following are the two main approaches available to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

# Equivalence Class Partitioning

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

- The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined.

For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e.,  $[1,10]$ ), then the invalid equivalence classes are  $[-\infty,0]$ ,  $[11,+\infty]$ .

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined.

For example, if the valid equivalence classes are  $\{A,B,C\}$ , then the invalid equivalence class is  $\square - \{A,B,C\}$ , where  $\square$  is the universe of possible input values.

# 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.
- 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  $\leq$ , or conversely  $\leq$  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 example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is  $\{0,1,10,11\}$ .



# WHITE-BOX TESTING

- White-box testing is an important type of unit testing. A large number of white-box testing strategies exist.
- Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic.
- We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

# Basic Concepts

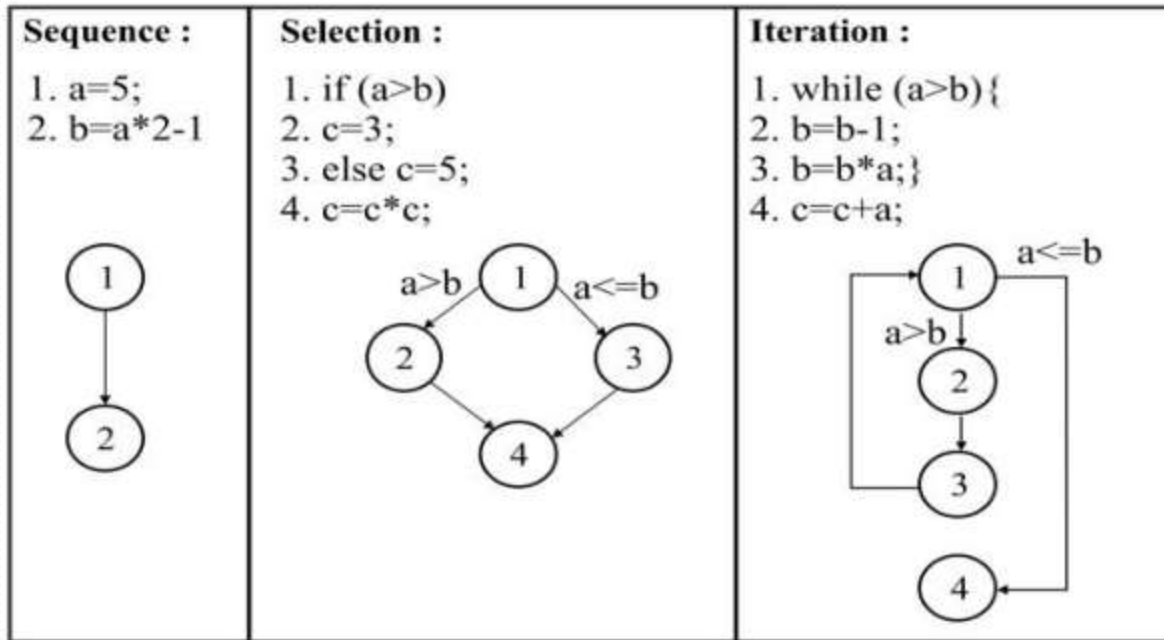
- A white-box testing strategy can either be coverage-based or fault based.
- **Fault-based testing**
  - A fault-based testing strategy targets to detect certain types of faults.
  - These faults that a test strategy focuses on constitutes the fault model of the strategy.
  - An example of a fault-based strategy is mutation testing.
- **Coverage-based testing**
  - A coverage-based testing strategy attempts to execute (or cover) certain elements of a program.
  - Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

## • **Testing criterion for coverage-based testing**

- A coverage-based testing strategy typically targets to execute certain program elements for discovering failures.
- The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.
- For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage.
- We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

# • Stronger versus weaker testing

- It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults.
- We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.
- A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.



- The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by A.
- On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.
- If a stronger testing has been performed, then a weaker testing need not be carried out.
- A test suite should, however, be enriched by using various complementary testing strategies.

## ● **Statement Coverage**

- The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.
- The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc.
- It can however be pointed out that a weakness of the statement-coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values.
- Never the less, statement coverage is a very intuitive and appealing testing technique.
- In the following, we illustrate a test suite that achieves statement coverage.

## • **Branch Coverage**

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn.
- For branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
- Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

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

- Proof: We need to show that

- (a) branch coverage ensures statement coverage,
- (b) statement coverage does not ensure branch coverage.

(a) Branch testing would guarantee statement coverage since every statement must belong to some branch

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

Consider the following code, and the test suite {5}.

- `if(x>2) x+=1;`
- The test suite would achieve statement coverage.
- It does not achieve branch coverage, since the condition  $(x > 2)$  is not made false by any test case in the suite.



## • Multiple Condition Coverage

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values.
- For example, consider the composite conditional expression  $((c1 \text{ .and.} c2) \text{ .or.} c3)$ .
- A test suite would achieve MC coverage, if all the component conditions  $c1$ ,  $c2$  and  $c3$  are each made to assume both true and false values.
- It is easy to prove that condition testing is a stronger testing strategy than branch testing.
- Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions.
- Therefore, multiple condition coverage-based testing technique is practical only if  $n$  (the number of conditions) is small.

## ● **Path Coverage**

- A test suite achieves path coverage if it executes each linearly independent paths at least once.
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

## ● **Control flow graph (CFG)**

- A control flow graph describes how the control flows through the program.
- We can define a control flow graph as the following:
  - A control flow graph describes the sequence in which the different instructions of a program get executed.
  - In order to draw the control flow graph of a program, we need to first number all the statements of a program.
  - There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

## • McCabe's Cyclomatic Complexity Metric

- McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program
- For structured programs, the results computed by all the three methods are guaranteed to agree.
- **Method 1: Given a control flow graph  $G$  of a program, the cyclomatic complexity  $V(G)$  can be computed as:**

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

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

- **Method 2:** An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows
- The cyclomatic complexity  $V(G)$  for a graph  $G$  is given by the following expression:
  - $V(G) = \text{Total number of non-overlapping bounded areas} + 1$
- In the control flow graph  $G$ , any region enclosed by nodes and edges can be called as a bounded area.
- This is an easy way to determine the McCabe's cyclomatic complexity.
- It can be shown that control flow representation of structured programs always yields planar graphs.
- But, presence of GOTO's can easily add intersecting edges.
- Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity does not apply.
- The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability of a program.
- **Method 3:** The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the
- program.
- If  $N$  is the number of decision and loop statements of a program, then the McCabe's metric is equal to  $N + 1$ .

## ● **Steps to carry out path coverage-based testing**

- The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.
2. Determine the McCabe's metric  $V(G)$ .
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. repeat
  - Test using a randomly designed set of test cases.
  - Perform dynamic analysis to check the path coverage achieved.

## • **Data Flow-based Testing**

- Data flow based testing method selects test paths of a program according to the definitions and uses of different variables in a program.
- All use criterion requires that all uses of a definition should be covered.
- Clearly, all-uses criterion is stronger than all-definitions criterion.
- An even stronger criterion is all definition-use-paths criterion, which requires the coverage of all possible definition-use paths that either are cycle-free or have only simple cycles.
- Data flow testing strategies are especially useful for testing programs containing nested if and loop statements.

## ● **Mutation Testing**

- Mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program.
- In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies.
- After the initial testing is complete, mutation testing can be taken up.
- The idea behind mutation testing is to make a few arbitrary changes to a program at a time.
- Each time the program is changed, it is called a mutated program and the change effected is called a mutant.
- A mutation operator makes specific changes to a program.
- For example, one mutation operator may randomly delete a program statement.
- A mutant may or may not cause an error in the program.
- If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.