

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

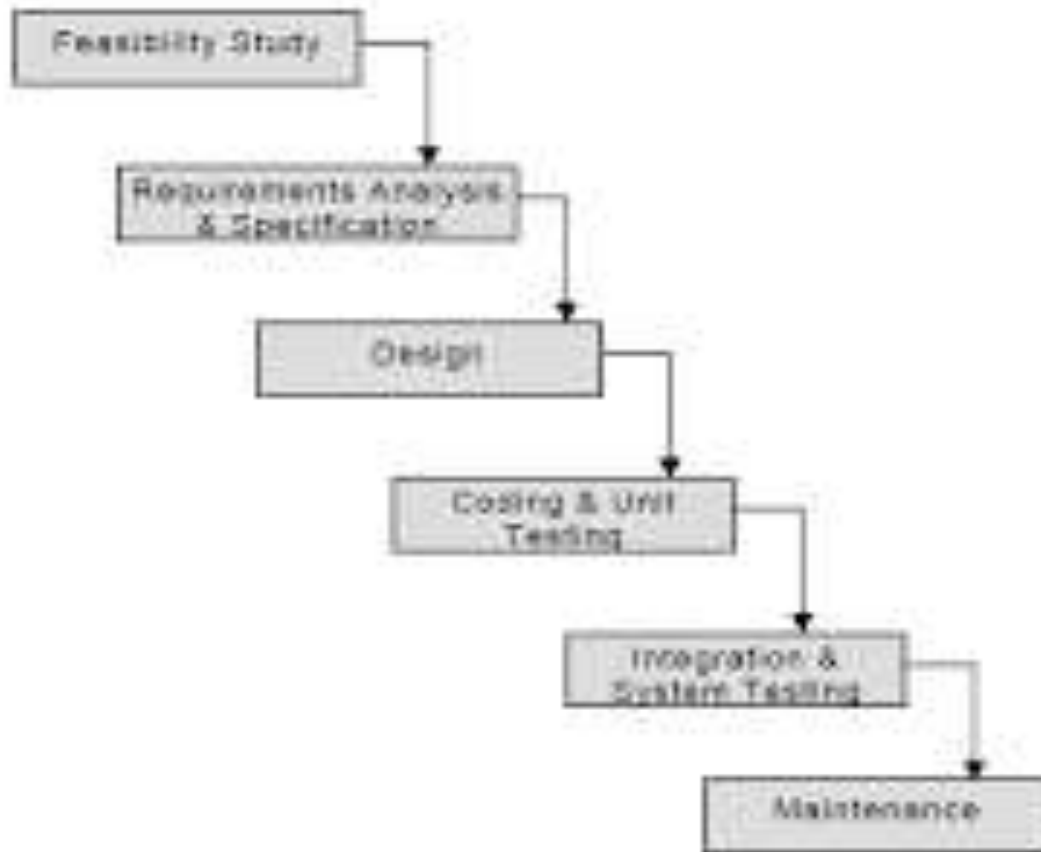


Fig 2.1: Classical Waterfall Model

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.
 - α -testing - testing performed by the development team
 - β -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.

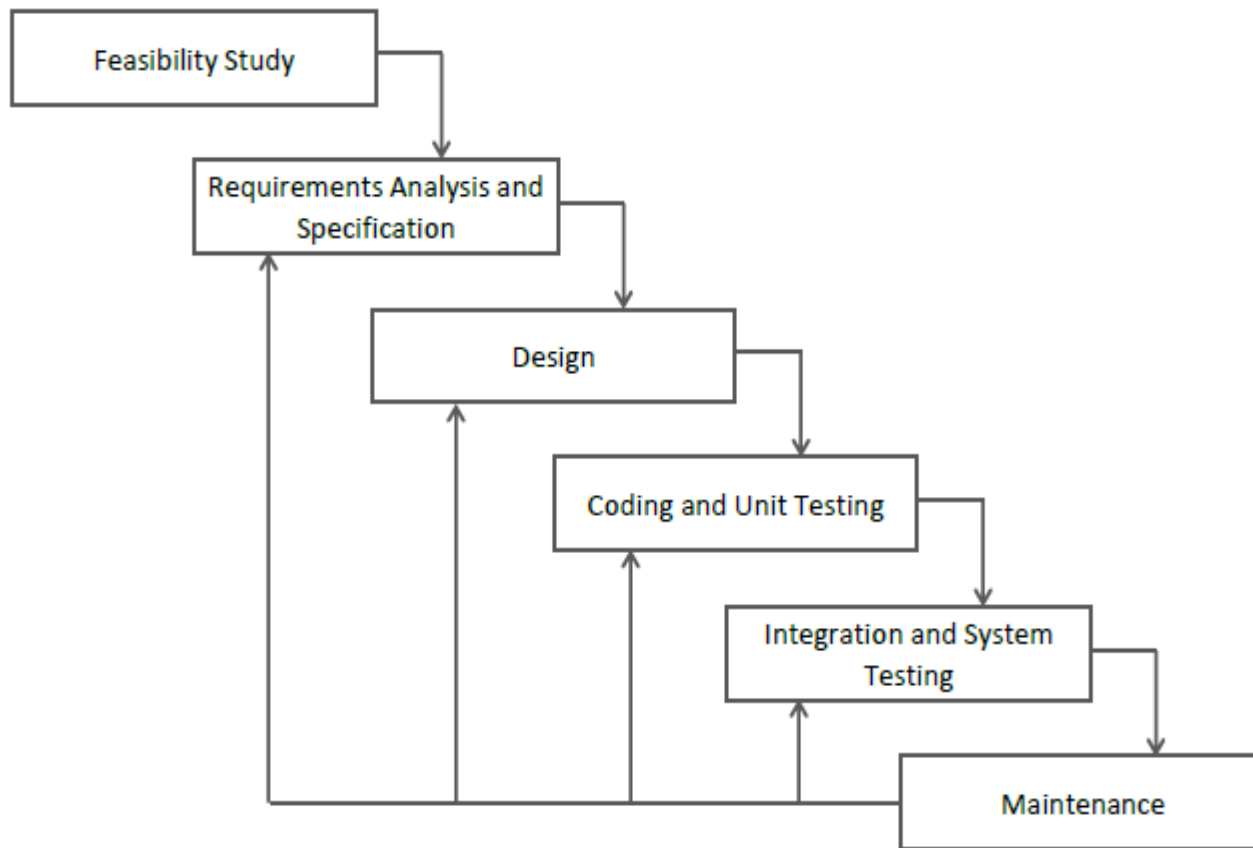


Figure 2: Iterative Waterfall Model

- 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.

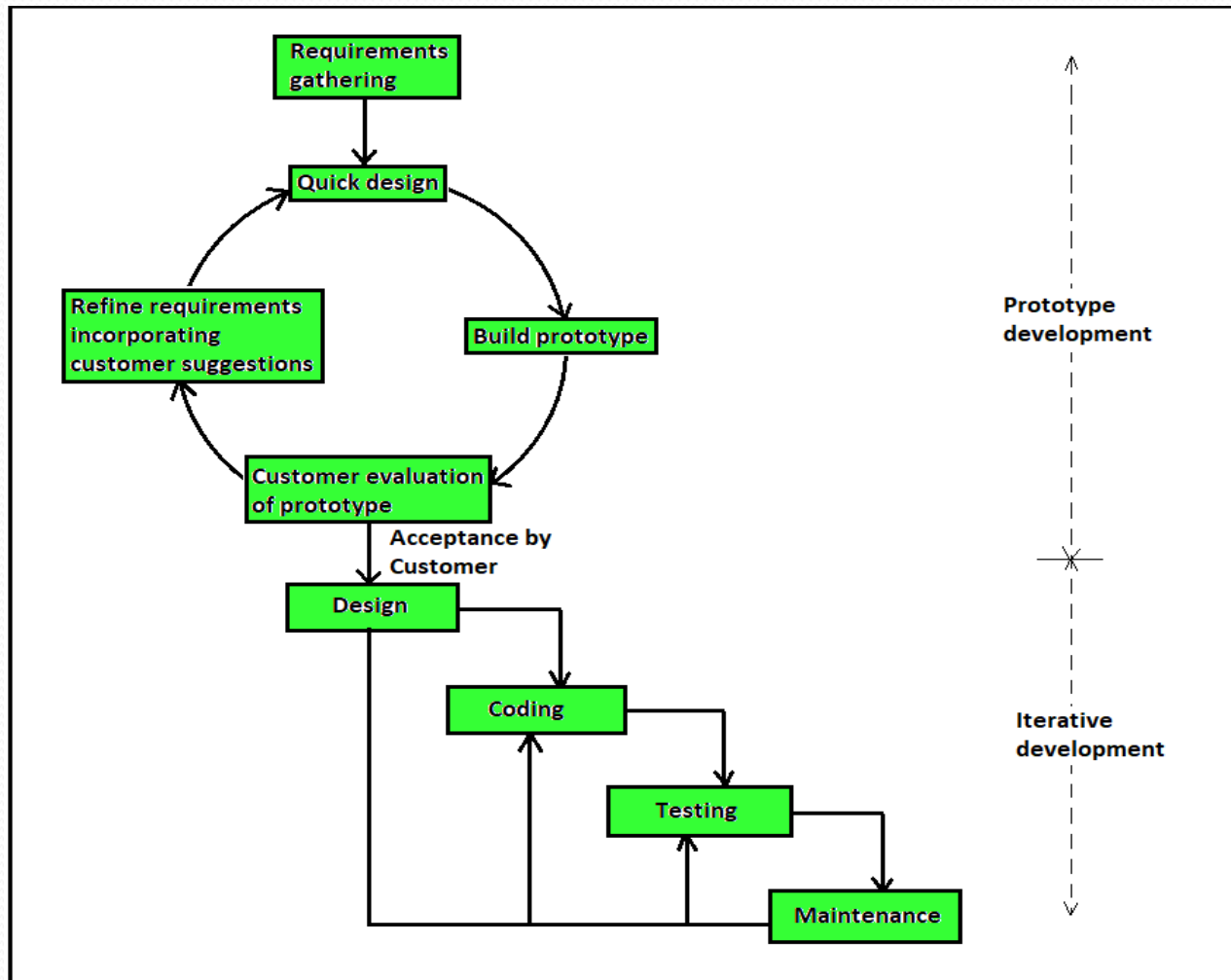
• Necessity of the prototyping model

- 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:**

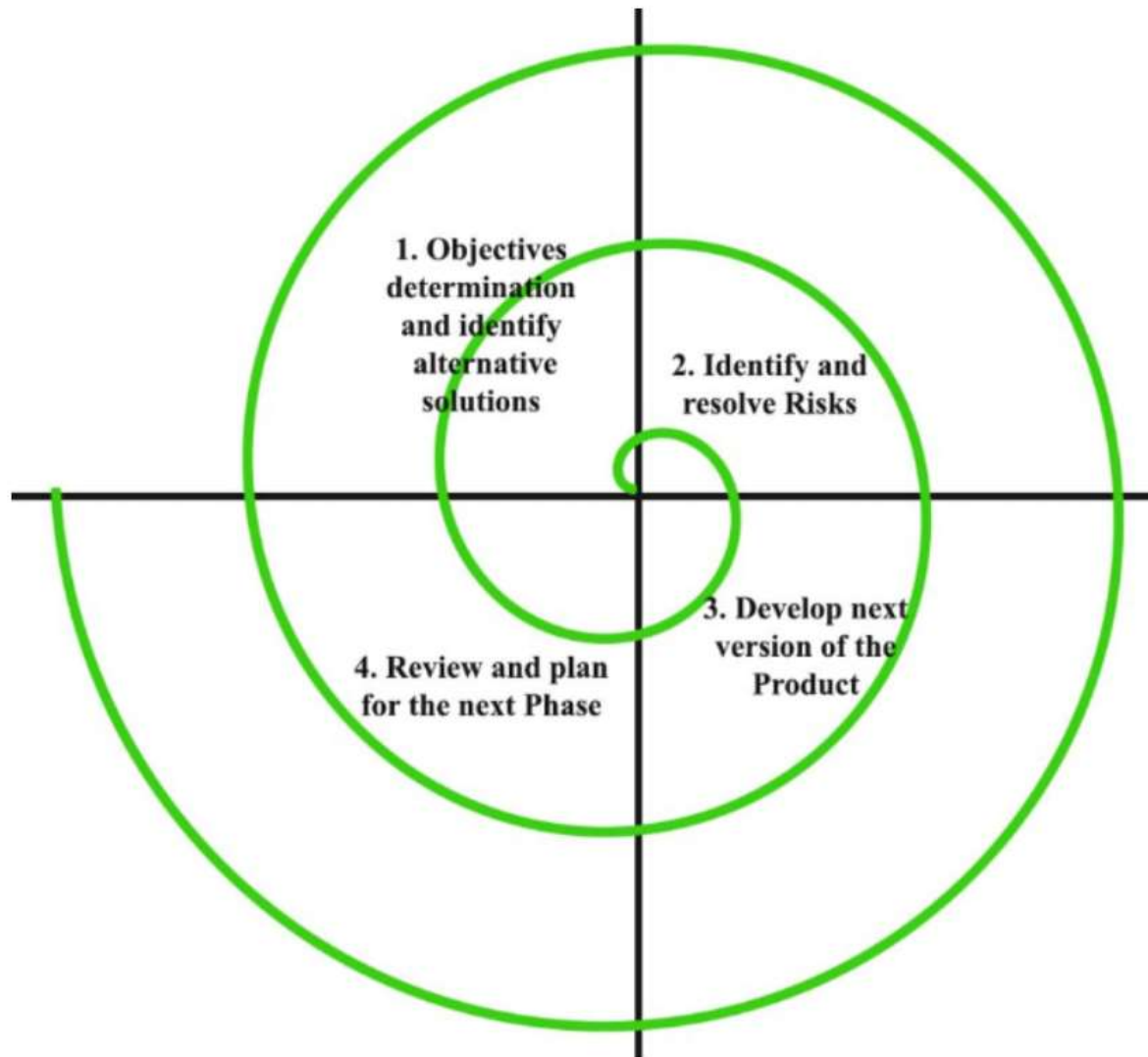
- If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts.
- In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.

- **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.