UNIT 1 : UNDERSTAND THE BASICS OF SOFTWARE ENGINEERING DESIGNS & LIFE CYCLE MODELS

INTRODUCTION TO SOFTWARE ENGINEERING

Software: A Text book description of software might be "Instructions that when executed provide desired features, function, and performance". To understand what software means, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Software has characteristics that are considerably different than those of hardware.

1. Software is developed or engineered; it is not manufactured in the classical sense.
2. Software doesn't wear out.
3. Although the industry is moving towards component based construction, most software continues to be custom built.

Software Application domains: Seven broad categories of software present continuing challenges for software engineers.
System Software, Application Software, Engineering / Scientific software, Embedded software, Product-Line software, Web applications, Artificial Intelligence software

Software Engineering: In order to build software that is ready to meet the challenges of the twenty-first century. You must recognize a few simple realities.

- A concerted effort should be made to understand the problem before a software solution is developed.
- Design becomes a pivotal activity.
- Software should exhibit high quality.
- Software should be maintainable

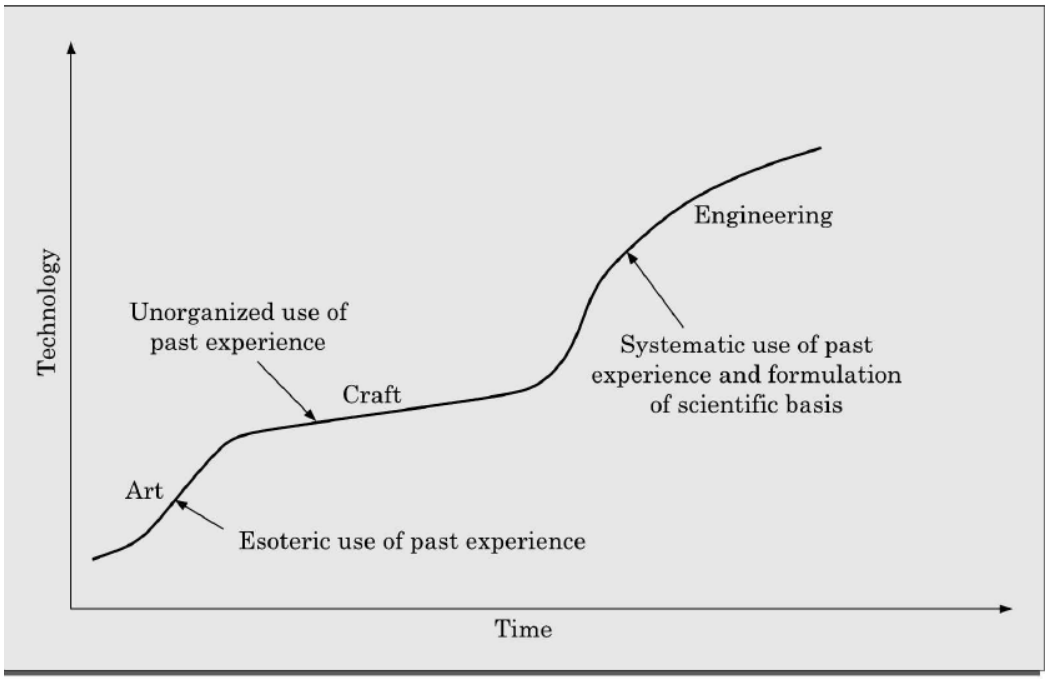
These simple realities lead to one conclusion: software in all of its forms and across all of its application domains should be engineered.
A definition proposed by Fritz Bauer
"Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

IEEE has developed a more comprehensive definition
"Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

1.1 KNOW THE EVOLUTION AND IMPACT OF THE SOFTWARE ENGINEERING

1.1.1 Evolution of an Art to an Engineering Discipline:

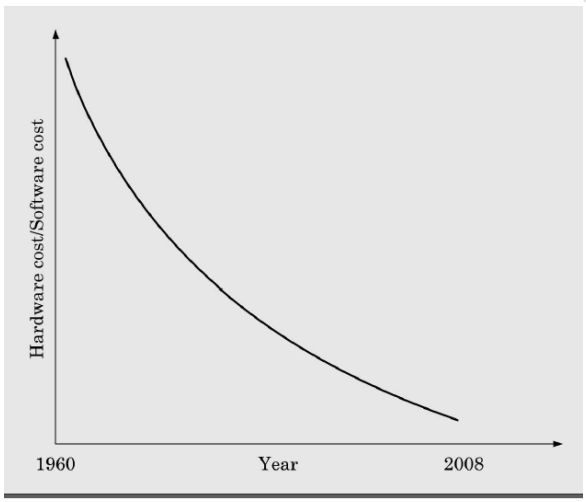
Software Engineering principles have evolved over the last sixty years. The early programmers used Built and Fix programming style. In built and fix style, normally a poor quality program is quickly developed with any plan or design. We can consider Built and Fix style as an art-since it is guided by intuition.
Programmers in modern software industry develop software by applying some well understood software engineering principles.



Evolution of technology with time.

We can analyze the evolution of software development styles over the last fifty years, we can easily notice that it has evolved from an mysterious art from to a craft form, and then has slowly emerged as an engineering discipline. In early days programmers know certain principles (or tricks) that they don't share with other programmers. Over the years all such good principles along with research have been organized into a form of knowledge that forms the discipline of software engineering.

1.1.2 A Solution to the Software Crisis?



Relative changes of hardware and software costs over time.

Organizations are spending larger and larger portions of their budget on software as compared to hardware. There are many factors for increase of software costs, the important one being: rapidly increasing problem sizes, lack of adequate training in software engineering techniques, increasing skill shortage, and low productivity improvements. The trend of increasing software costs is symptom of the present software crisis.

The solution to the present software crisis can come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

1.2 KNOW THE DIFFERENCE BETWEEN PROGRAMS AND SOFTWARE PRODUCTS

Programs are developed by individuals for their personal use. They are, small in size and have limited functionality, usually lack good user interface and proper documentation. For example the programs developed by a student as part of assignment are programs and not software products.

Software products have multiple users and have good user interface, proper user manual and good documentation. Since a software product has a large number of users, it is systematically designed, carefully implemented and thoroughly tested.

1.3 UNDERSTAND THE EVOLUTION OF SOFTWARE ENGINEERING DESIGN

Software engineering has evolved over many years in the past. This evolution is the result of a series of innovations experience of writing good programs. Let us examine a few of these programming experiences which have contributed to the development of the software engineering discipline.

1.3.1 Early Computer Programming

Early commercial computers are very slow and too elementary as compared to today's. Those programs were usually written in assembly language. Program lengths were typically limited to a few hundred lines of assembly code. Programmers wrote programs without any proper strategy, plan or design. We can designate this style of programming as build and fix.

1.3.2 High Level Language Programming

Computers became faster with the introduction of the semiconductor technology with the availability of more powerful computers; it became possible to solve large and more complex problems. At this time High-Level languages such as FORTRAN, ALGOL and COBOL were introduced. However programmers were still using Built and fix style of software development. Programs were limited to around few thousand lines of code.

1.3.3 Control Flow-Based Design

As the size and complexity of program keep on increasing, the built and fix style proved to be insufficient. To cope up with the situation experienced programmers advised other programmers to pay particular attention to the design of a programs control flow structure.

"A Programs control flow structure indicates the sequence in which the programs instructions are executed"

In order to help develop programs having good control flow structure, the flow charting technique was developed.

A logical extension: Structured Programming

It was proven by Bohm and Jacopini that only three programming constructs: Sequences, Selection and Iteration where sufficient to express any programming login. Gradually everyone accepted that it is indeed possible to solve any programming problem without using GOTO statement and indiscriminate use of GOTO statement should be avoided. This formed the basis for the **Structured Programming** methodology

A program is called structured when it uses only the sequences, selection, and iteration types of constructs and is modular. Very soon several programming languages such as PASCAL, MODULA, C etc become available which support structured programming.

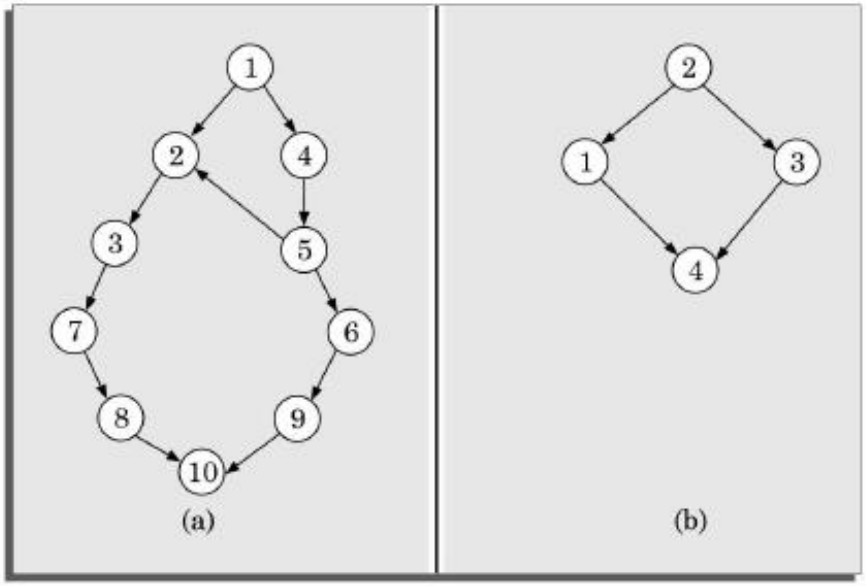
```
1       if(customer_savings_balance>withdrawal_request) {
2   100:    issue_money=TRUE;
3           GOTO 110;
        }
4       else if(privileged_customer==TRUE)
5           GOTO 100;
6       else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8       GOTO 130;
9   120:    print(error);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

(a) An example unstructured program

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

(b) Corresponding structured program

Two programs.



Control flow graphs of the programs in Figures (a) and (b).

1.3.4 Data Structure-Oriented Design:

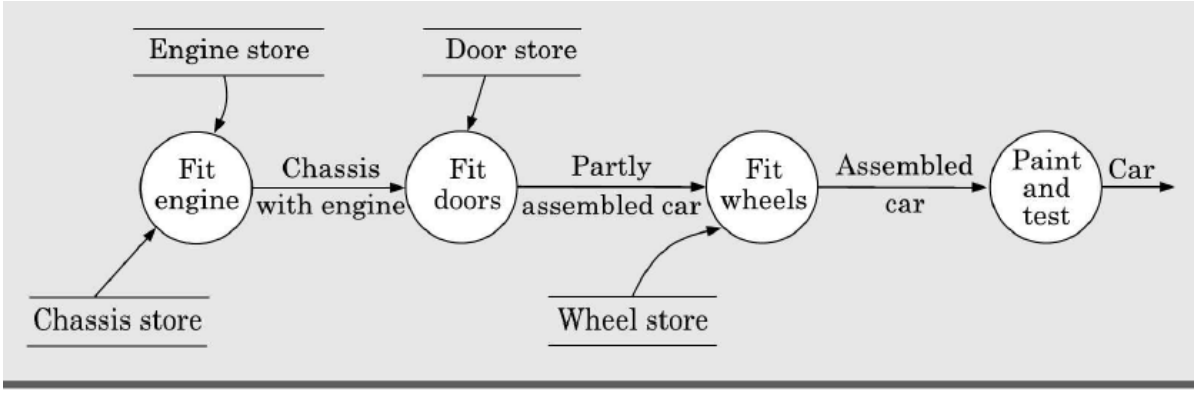
Computers become more powerful with the use of ICs. These could now be used to solve more complex problems. Using Data structure oriented Techniques, first a programs data structures are designed. In the next step, the program design is derived from the data structure. An example of Data structure oriented design technique is the JSP (Jackson's Structured Programming) beveloped by Michael Jackon.

1.3.5 Data Flow-Oriented Design

Computers become still faster with the introduction of VLSI. Therefore, software engineers looked out for more effective techniques for designing software products and soon data flow – oriented techniques were proposed.

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

The functions and the data items that are exchanged between the different functions are represented in a diagram known as Data Flow Diagram (DFD). The program structures can be designed from the DFD representation of the problem.



Data flow model of a car assembly plant.

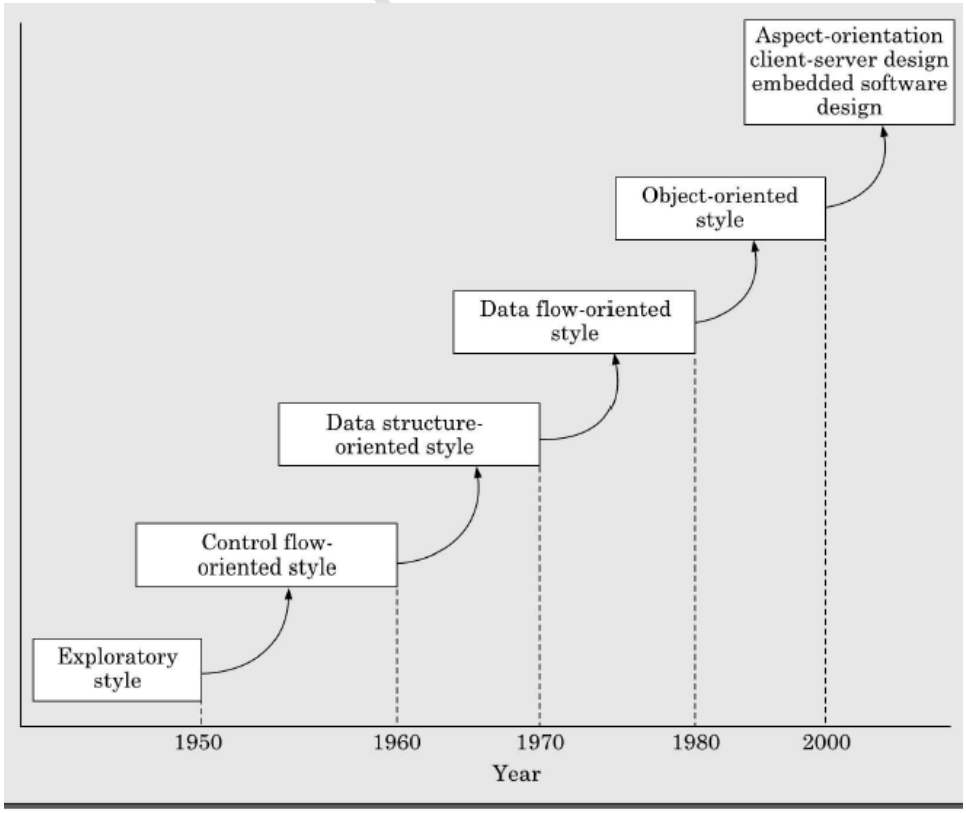
1.3.6 Object oriented design

The Data flow oriented techniques evolved into Object-Oriented Design (OOD) Technique. Object-Oriented design is an naturally approach where the natural objects relevant to a problem are first identified and then the relationship among the objects such as inheritance, references are determined.

Each object acts as a data hiding entity. Object oriented techniques have gained wide spread acceptance because of their simplicity, the scope of code and design reuse, promise of lower development time, lower development cost, more robust code, easier maintenance.

1.3.7 Other Developments

In addition to the advancements made to the software design techniques, several other new concepts and techniques for effective software development were introduced. These new techniques include life cycle models, specifications techniques, project management techniques, testing techniques, debugging techniques, quality assurance techniques, software measurement techniques, computer-aided software engineering (CASE) tools etc., The development of these techniques accelerated the growth of software engineering as a discipline.



Evolution of software design techniques.

1.4 Explain Software Life Cycle Models:

A life cycle model prescribes the different activities that need to be carried out to develop a software product and the sequencing of these activities.

The software life cycle is also sometimes referred to as the System Development Life Cycle(SDLC).

A software life cycle is a series of identifiable stages that a software product undergoes during its lifetime.

The software Life cycle can be considered as business process for software development, and therefore, a software life cycle is also often referred to as a software process.

The first stage in the life cycle of any software product is usually the feasibility study stage. The subsequent stages are requirement analysis, design, coding, testing and maintenance. Each of this stages is called a life cycle phases. A life cycle model graphically represents the different phases of a life cycle and their ordering accompanied by with textual representation.

Why use a Life Cycle Model:
Software Development Organizations have realized that adherence(strictly follow) to a suitable well-defined life cycle model helps to produce good quality products and that too without time and cost overruns.

Why document a Life cycle Model?
Good software organizations normally document all the information regarding the outputs to be produced at the end of different phases, methodologies to be adopted etc into a framework called the **Organizations software development model**. The organizations expect the fresh engineers recruited to first master the organizations software development model.

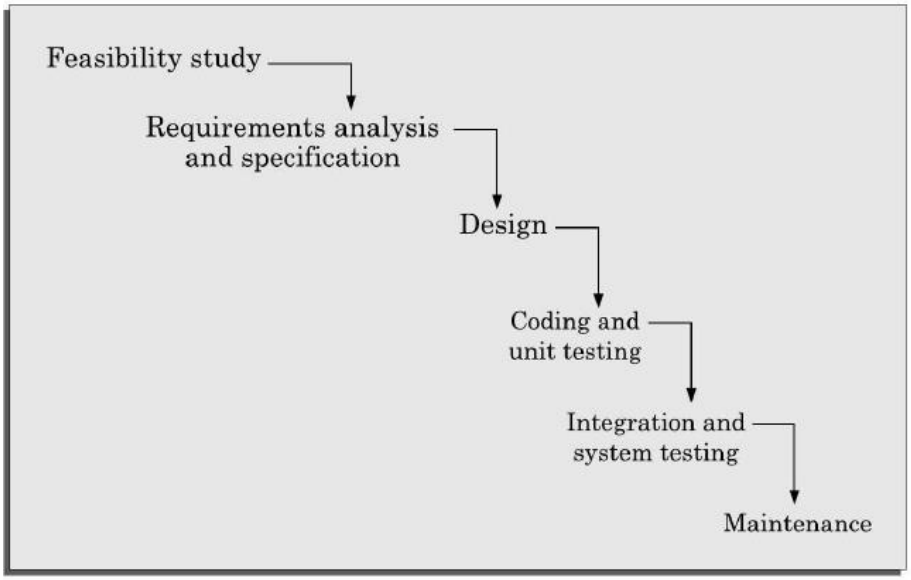
1.4.1 Classical Waterfall Model:

All Life cycle models are in some way or other based on the Classical Waterfall Model. We therefore need to first understand the classical waterfall model well, to be able to appreciate and develop proper understanding of other life cycle models.

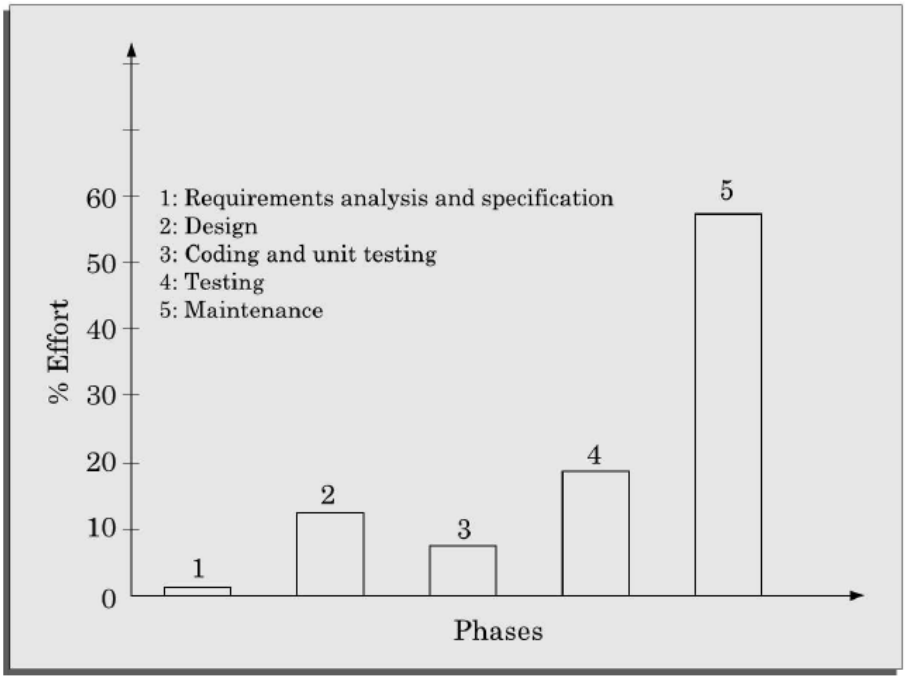
The classical waterfall model divides the life cycle into the phases as shown in figure. The diagrammatic representation of this model resembles a cascade of waterfalls. This justifies the name of the model.

Phases of Classical Waterfall Model: The waterfall model breaks down the life cycle into set of phases. The different phases of this model are feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing and maintenance. The phases feasibility study to the integration and system testing phase are known as development phases. The maintenance phase commences after completion of the development phase. An activity that spans all phases of any software development is project management.
The amount of effort necessary for completing the activities of different phases for a typical product is shown in figure.

On the average about 60% of the total life cycle effort is spent on the maintenance activities alone.



1) Feasibility Study: The main aim of the feasibility study activity is to determine whether it would be financially and technically feasible to develop the product. The feasibility study activity involves analysis of the problem and collect of all relevant information relating to the product. The collected data are analyzed to arrive at the following



Relative effort distribution among different phases of a typical product.

1. **An Abstract problem specification:** Only important requirements of the customer are captured details of the requirements are ignored.
2. **Formulation of the different strategies for solving the problem:** All different ways in which the problem can be solved is identified.
3. **Evaluation of the different solution strategies:** The different solution strategies are analyzed to examine their benefits and shortcomings.

During the feasibility study very high level decisions regarding the exact solution to be adopted are made. At this stage it may also be determined that none of the solution if feasible due to high cost, resource constraints or some technical reasons and the project may be abandoned.

2. Requirements Analysis and Specification: The aim of this phase is to understand the exact requirements of the customer and to document them properly.

Requirement gathering and Analysis: The goal of requirement gathering is to collect all relevant information regarding the product to be developed from the customer with a view to clearly understand the customer requirements. The goal of requirement analysis is to weed out the incompleteness and inconsistencies in their requirements.

Requirements Specification: The customer requirements identified during the requirement gathering and analysis activity are organized into a Software Requirements Specification (SRS) document. The three important contents of this document are the Functional Requirements, Non-Functional Requirements and the goals of Implementation. Functional requirements describe functions to be supported by system. Each function can be characterized by their input, processing and output to be produced. The non-Functional requirements identifies the performance requirements, the required standards to be followed etc.,

- The SRS document is written using end user Terminology
- SRS is understandable by customer.
- SRS is reviewed and approved by customer.
- SRS serves as contract between development team and customer
- Disputes between customer and development can be settled by examining SRS
-

3. Design: The goal of the design phase is to transform the requirements specified in the SRS into a structure that is suitable for implementation in some programming language. In Technical Terms during the design phase the software architecture is derived from SRS. Two different design approaches are used at present.

1. Traditional Design Approach: Used by many software development houses it is based on Data-Flow-oriented Design Approach. Traditional Design Approach consists of structured Analysis and Structured Design Activities.

2. Object-Oriented Design Approach: In this technique various objects that occur in problem domain and solution domain are first identified and relationship that exist among these objects are indentified. The OOD approach has several advantages such as lower development time and effort and better maintainability of the product.

4. Coding and Unit Testing: The coding phase is also sometimes called the implementation phase since the design is implemented into a workable solution. Each component of the design is implemented as a program module.
After coding is complete, each module is unit tested. Unit Testing involves testing each module in isolation from other modules, then debugging and documenting it. The main objective of unit testing is to determine the correct working of the individual module.

5. Integration and system Testing: Integration of different modules is undertaken once they have been coded and unit tested. Integration of various modules are normally carried out incrementally over a number of steps. During each integration step previously planned modules are added to the partially integrated system and resultant system is tested.
Finally after all the modules are integrated and tested, system testing is carried out. The goal of system testing is to ensure that the development system conforms to its requirements laid out in the SRS document.
System testing consists of 3 different kinds of testing activities.

1. **α-testing:** α testing is the system testing performed by the development team.
2. **β-testing:** This is the system testing performed by a friendly set of customers.
3. **Acceptance Testing:** This is the system testing performed by the customer himself after product delivery to determine whether to accept the delivered products or to reject it.

6. Maintenance: Maintenance involves performing any one or more of the following three kinds of activities.
1. Corrective Maintenance: This type of maintenance involves correcting errors that were not discovered during the product development phase.
2. Perfective maintenance: This type of maintenance involves enhancing the functionalities of the system according to the customer's requirements.
3. Adaptive Maintenance: This type of maintenance usually required for porting the software to work in new environment such as new computer platform or with new operating system.

Shortcomings of the Classical Waterfall Model:

1. The classical waterfall model is an idealistic one since it assumes that no error is ever committed by the engineer during any of the life cycle phases, and therefore leaves no scope for error correction.
2. This model assumes that all requirements are defined correctly at the beginning of the project in real projects customers keep on changing requirements as the development proceeds. In waterfall model it becomes difficult to accommodate later requirements change.
3. Classical Waterfall model assumes that all phases are sequential. However that is rarely the case.

Irrespective of the life cycle model actually followed for the product development, the final document is written to reflect a classical waterfall model of development.

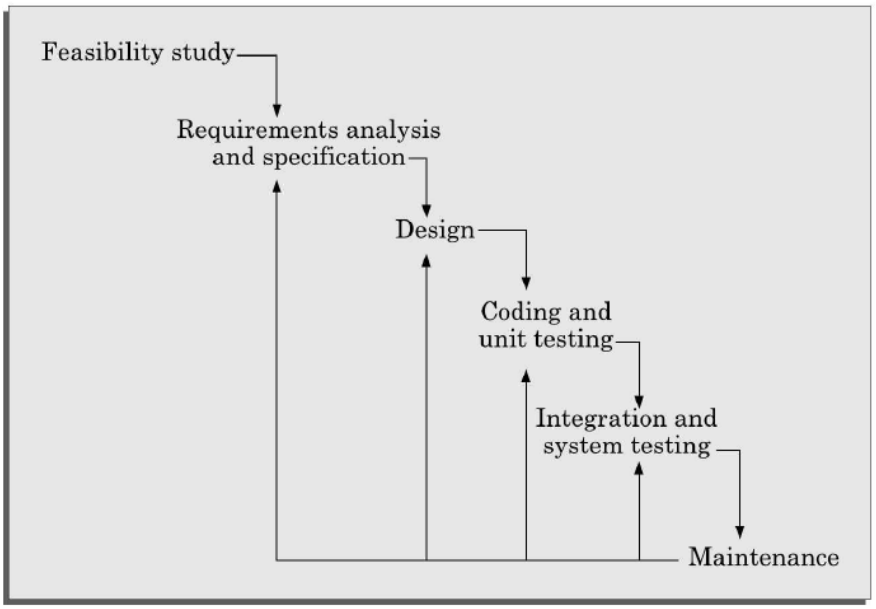
1.4.2 Iterative Water fall Model In practical software development work, it is not possible to strictly follow the classical waterfall model. We branded the classical waterfall model as an idealistic model.
We can view the Iterative Waterfall model as making necessary changes to the classical waterfall model so that it becomes applicable to practical software development projects. The main change to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phase. The feedback path allows for correction of the errors committed during a phase, as and when these are detected in a later phase. However there us no feedback path to the feasibility stage. This means that the feasibility study error cannot be corrected.

Phase containment of Errors: Errors are expected in almost every phase of development, it is desirable to detect these errors in the same phase in which they occur. This can reduce the effort

required for correcting bugs. For example if a design problem is detected in the design phase itself, this problem can be taken care of much more easily than if the error is identified at the end of system testing phase.

"The principle of detecting errors as close to their point of introduction as possible is known as *Phase containment of Errors* This is an important software engineering principle.

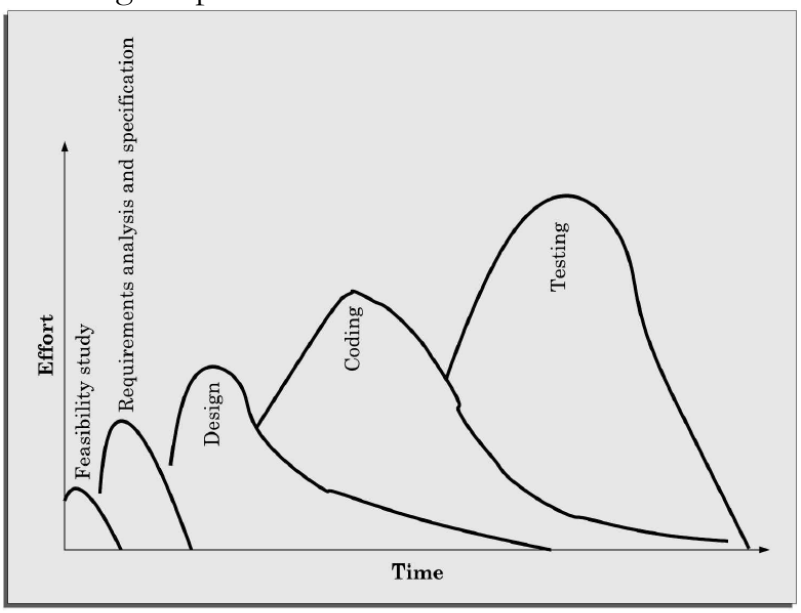
How can phase containment of Errors be achieved? An important technique that is frequently used this regard is to conduct reviews after every milestone.



Iterative waterfall model.

In spite of best effort to detect errors in the same phase still some errors can escape detection and would be detected in later phase. It includes the rework required to the phase later due to errors of that phase detected at the later phases. This makes different life cycle phases overlap in time.

Another reason for phase overlap is that once a developer completes his assigned work for a phase, he proceeds with his activities in the subsequent phase rather than idling till the other developers complete the assigned phase.



Distribution of effort over time for various phases in the iterative waterfall model.

Shortcomings of the Iterative Waterfall Model:

A good understanding of waterfall model is necessary to appreciate and understand the other development processes. However waterfall model suffers from many shortfalls.

1. The waterfall model cannot satisfactorily handle the different types of risks that a real life software project may suffer from. For example, waterfall model assumes that requirements are completely specified before development. It cannot be used in projects where customer is not clear about his requirements and can provide rough requirements only.
2. To achieve better efficiency and high productivity, most real life projects find it difficult to follow the rigid phase sequence of waterfall model. Rigid phase sequence means that a phase can start only after the previous phase is completed in all respects. It would create blocking states.

1.4.3 PROTOTYPING MODEL

The prototyping model requires that before carrying out the development of the actual software, a working prototype of the system should be built. A prototype is usually built using several short cuts. The short cuts might involve using inefficient, inaccurate or dummy functions.

For many projects, there are several advantages of building a prototype before the actual product is developed.

1. An important use of a prototype can be to illustrate the input data formats, messages, reports, interactive dialogues to the customer. This can be a valuable mechanism for gaining better understanding of the customers' needs.
2. Another important situation where the prototyping model can be useful where the exact technical solutions to be adopted are unclear to the development team. A prototype would help them to examine technical issues associated with the product development. (for example developing a prototype compiler for a very small language leads to understating the issues associated with writing a compiler for a command language)
3. It is impossible to get it right the first time, and one must plan to throw away the first product in order to develop a good product later.

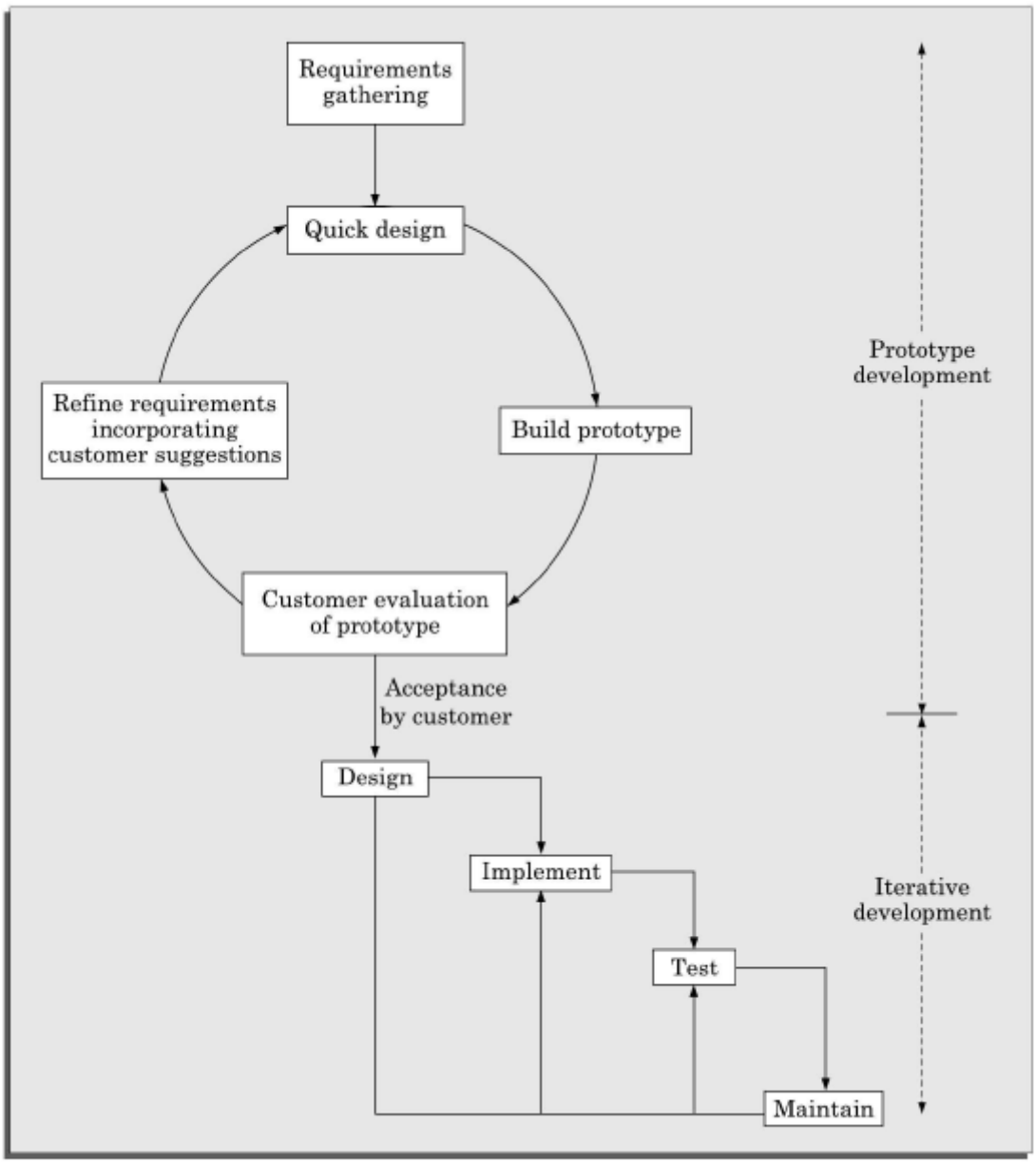
The first phase is prototype development followed by an iterative development cycle. In this model, prototyping starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for his

evaluation. Based on customer feedback the requirements are refined and the prototype is modified. Once the customer approves the prototype the actual system is developed using interactive waterfall model.

In spite of availability of working prototype, the SRS document is required to be developed for traceability analysis, verification, and Test case design.

The code for prototype is usually thrown away, even though the construction of a working prototype might involve additional cost, The overall development cost might turn out to be low in prototype model.

By constructing the prototype many customer requirements get properly defined. This minimizes the change requests from the customer and the associated redesign cost.



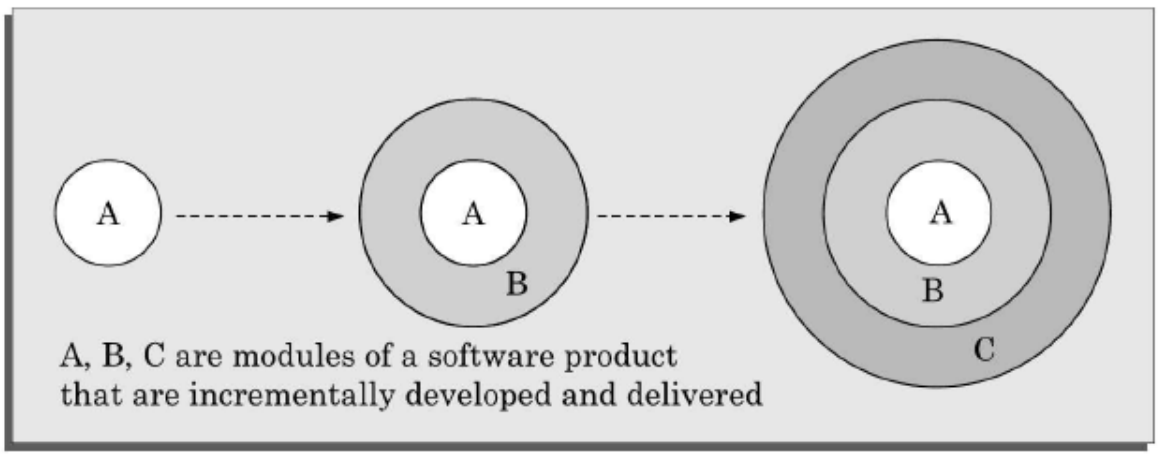
Prototyping model of software development.

1.4.4 EVOLUTIONARY MODEL:

This life cycle model is also referred to as the **successive versions** model and sometimes as the **incremental model**. The evolutionary software development process is sometimes referred to as **design a little, built a little, test a little deploy a little model**.

Life cycle Activities : In evolutionary life cycle model, the software requirements is first broken down into several modules(or functional units) that can be incrementally constructed and delivered.

The development team first develops the core modules of the system. The core modules are those that do not need services from the other modules. On the other hand non-core modules need services from the core modules. This initial product skeleton is refined into increasing levels of capability by adding new functionalities in successive versions. Each evolutionary version may be developed using iterative waterfall model.



Evolutionary development of a software product.

The evolutionary model is shown in figure. Each successive version is fully functioning software capable of performing more work than the previous versions.

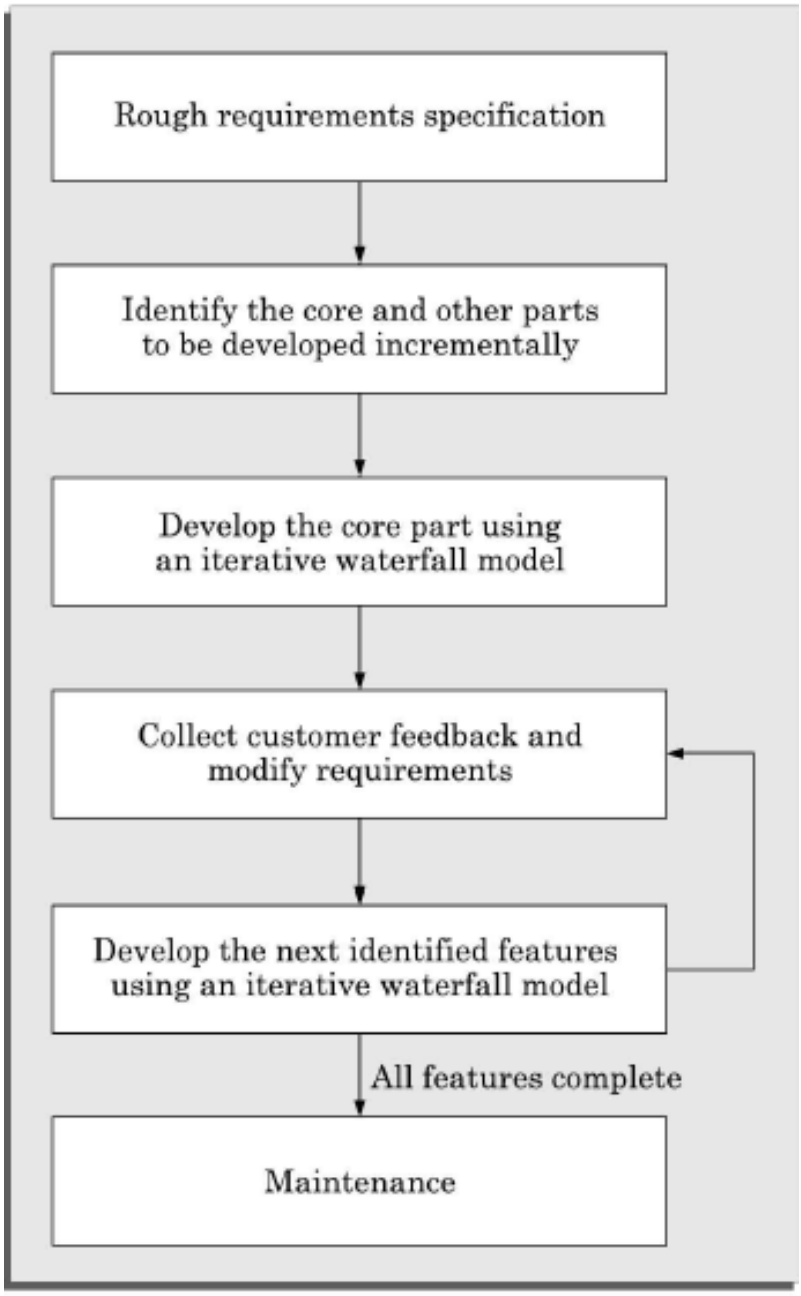
Advantages:

1. In this model the users gets chance to experiment with partially developed software much before the complete version of the system is released.
2. The evolutionary model helps to accurately extract user requirements during the delivery of different versions of the software. As a result the change requests after delivery of the complete software become very less.

Disadvantages: The main disadvantages of the successive versions model are that for most practical problems it is difficult to divide the problem into several versions.

Types of projects for which suitable

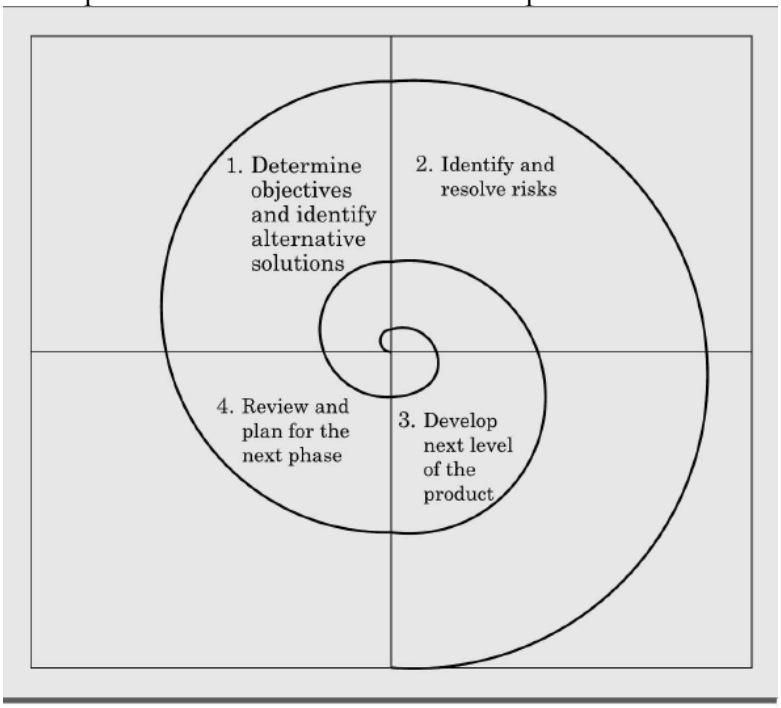
- The evolutionary model is normally useful for very large products.
- It is used when the customer prefers to receive the product in increments.
- The evolutionary model is a very natural model to use in Object-oriented software development projects. Objects are more or less self contained units that can be developed independently.



Evolutionary model of software development.

1.4.5 SPIRAL MODEL

The Spiral model of software development is shown in figure.



Spiral model of software development.

The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops of the spiral is not fixed and can vary from product to product. Each loop of spiral is called a phase of the software process.

Over each loop one or more features of the product are elaborated and analyzed and the risk at that point of time are identified and are resolved through prototyping. Based on this the features are implemented.

Risk handling in Spiral Model: A risk is any adverse circumstance that might block the successful completion of the project. Example the risk involved in accessing data from remote database. The risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate.

Phases of the Spiral Model: Each phase in this model is split into four sectors (or quadrants). In the first quadrant, some features of the product are identified based on the severity of the risk and how crucial it is to the overall product development. Also alternative solutions possible for the phase under consideration are proposed.

Second quadrant, the alternative solutions are evaluated to select the best possible solution. Alternative solutions are evaluated by developing an appropriate prototype.

Activities during the third quadrant consist of developing and verifying the next level of the product.

Activities during the fourth quadrant concern reviewing the results of the stages traversed so far with the customer and planning the next iteration around the spiral.

Pros and Cons of Spiral Model: There are few disadvantages with spiral model.

1) The spiral model usually appears as a complex model to follow.
2) It requires knowledgeable staff.
3) It is not suitable for development of outsourced projects.

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.

Spiral model as a Meta Model The spiral model can be viewed as a Meta model, since it subsumes all the discussed models.

This enables the developers to understand and resolve the risks at each evolutionary level (i.e iteration along the spiral). The spiral model uses prototyping as a risk reduction mechanism and also retains the step wise approach of the waterfall model.

1.4.6 COMPARISON OF DIFFERENT LIFE CYCLE MODELS

The classical waterfall model can be considered as the basic model and all other life cycle models as additions of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases. This problem may be overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. It is especially popular for development of the user-interface part of the projects.

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

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models

Another important advantage of the incremental model is that it reduces the customer's pain of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

Selecting an appropriate Life cycle Model for a Project is based on

1. Characteristics of the software to be developed.
2. Characteristics of the development team
3. Characteristics of the customer

Prepared By
N RajaSekhar, HCME –I Shift
AANM & VVRSR Polytechnic, Gudlavalleru