# SOFTWARE ENGINEERING

## Software Development Life Cycle { SDLC }

- Typical PHASES as per IEEE (institute of electrical and electronic engineering)
  - Feasibility study
    - T - Technical feasibility
    - E - Economic feasibility
    - L - Legal feasibility
    - O - Operational feasibility
    - S - Schedule feasibility
  - Analysis
  - Design
  - Coding
  - Testing
  - Delivery and Maintenance

## Waterfall Model

### Advantages

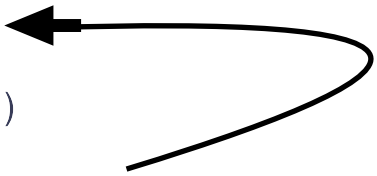
- Simple ( easy to understand )
- Structured approach ( less/no room for making mistakes )
- Works well with small projects
- Fully working design, documents are present in the beginning itself
- Suited for projects with **fixed** requirements and scope

### Disadvantages

- Requires defining all requirements at the beginning of the project
- Not suitable for accommodating change
- Working version of the system is not seen/available until late in the project life
- Not suitable for large projects
- Real projects are **rarely** sequential

## Prototyping Model

- Requirements
- Quick Design ( not accepted by customer )
- Implementation
- Customer



Evaluation

( accepted by customer )

- Design
- Implementation & Unit Testing
- Integration & System Testing
- Operation & Maintenance

Advantages

- Little chance of rejection ( customer is involved all through )
- Reduced risk of failure ( testing each prototype )
- Encourages innovation and flexible designing ( modify and discard prototypes )
- Prototypes offer training opportunities to future users

◉ Look in Dr. Yogesh Singh's book

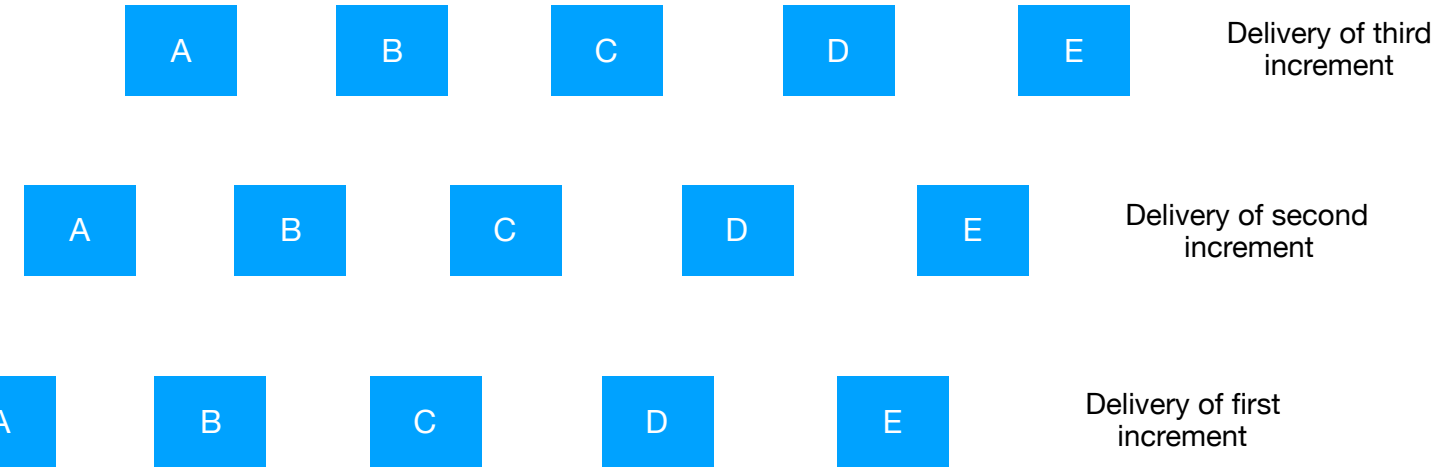
Disadvantages

- Time consuming
- Costly
- May encourage Excessive change requests causing difficulty
- A Disaster if final product is rejected/fails

## Incremental Model

- Requirement gathering

( the steps go on till the Nth increment until customer is satisfied )




- A - Communication
- B - Planning
- C - Modelling ( Analysis and Design )
- D - Construction ( Coding and Testing )
- E - Deployment ( Delivery and Feedback )

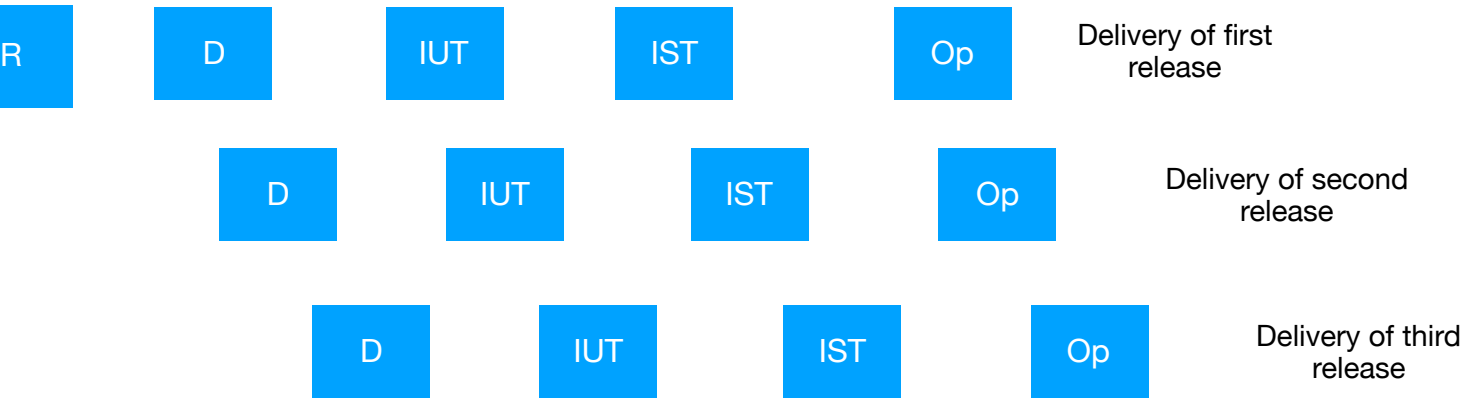
Thursday, 16th Jan 2020

## Incremental Model

- Applies the waterfall model incrementally
- The series of **releases** is referred to as **increments,** With each increment providing more functionality and features
- After the first increment, a core product is delivered which can be used by the customer
- Based on customer feedback, plan is developed for the next increments, and modifications are made accordingly
- This process continues with increments being delivered until the complete product is delivered

## Iterative Model

- This model has the same phases as the Waterfall Model, but with fewer restrictions
- A usable product is released at the end of each cycle, with each release providing additional functionality




( iterations until he complete product is delivered )

R => Requirements D => Design Op => Operations

IUT => Implementation and Unit Testing IST => Integration and Sytem Testing

## Spiral Model

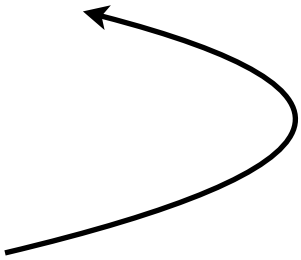
- Generalisation of above models
  ( All above models are special cases of Spiral model )
- The above models do NOT account for the Risk involved. There is ALWAYS some risk involved in each step in the development process
- **RISK DRIVEN** : The ONLY model that accounts for RISK
- Unlike many other software processing models, the spiral model incorporates **project risk factor**

I. **Determine objectives, alternative constraints**

II. **Evaluate alternatives; identify and resolve risks**

III. **Develop and verify next level product**

IV. **Plan next phases**



**Progress through the steps as many times as necessary**

**## photo of spiral with different steps**

- in Spiral model, one phase (cycle) is split roughly into four sectors of major activities
  - Planning : Determination of
    - objectives
    - alternatives
    - constraints
      - Financial
      - Technical
      - Time
  - Risk Analysis : Identify and resolve the risks involved
  - Development : Product development and testing
  - Assessment : customer evaluation

** **advantages and disadvantages of all the models**

## 29/01/2020

Typical steps for any software development model

- Requirement (analysis and specification)
- Design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

## take notes of unit II

Q: various steps involved in Requirements Engineering process

A:

Requirements elicitation

Requirements analysis

Requirements documentation

Requirements review/validation

---

SRS — user requirements - layman terms for user/client

— system requirements - technical for system builders

both required when user is not aware of adequate technical concepts

**Requirements**

- functional - what the system should do
- non-functional -

## Requirements Elicitation techniques

- ✦ Interviews
- ✦ Brainstorming session
- ✦ Facilitated Application Specification Technique (FAST)
- ✦ Quality Function Deployment
- ✦ The Use Case Approach

## Requirements Analysis

1. Draw the context diagrams
2. Development of a prototype (optional)
3. Model the requirements
4. Finalize the requirements

## Requirements Documentation

a.k.a. **Software Requirements Specification (SRS)**

### Nature of the SRS

1. Functionality
2. External Interfaces
3. Performance
4. Attributes
5. Design constraints imposed on an implementation

## Characteristics of a good SRS

1. Correct
2. Unambiguous
3. Complete
4. Consistent
5. Ranked for importance and/or stability
6. Verifiable
7. Modifiable
8. Traceable

30/01/2020

##only attendance##

03/02/2020

## Requirements Review/Validation

### 1 Requirement Review

### 2 Prototyping

## Unit III : SOFTWARE DESIGN

Q. What is Design ?

Highly significant phase in the Software Development where the designer plans **how** a software should be produced in order to make it functional, reliable and reasonable easy to understand, modify and maintain.

Design involves transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements (specified in SRS).

| DESIGNER | | |
|---|---|---|
| Conceptual Design | \| | Technical Design |
| (WHAT) | \| | (HOW) |
| customer | \| | developer |

Designers are intermediaries between customers and system builders. Designers prepare Conceptual Design that tells the customers exactly WHAT the system will do.

Once this is approved by the customer(s), the Conceptual Design is translated into a much more detailed document, the Technical Design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem.

Conceptual Design tells us WHAT the system will do, whereas Technical Design tells us HOW the system is to work.

Questions Conceptual Design answers :

* Where will the data come from?
* What will happen to the data in the system?
* What choices will be offered to users?
* What are the timings of events?
* How will the reports and screens look like?

(Non-exhaustive)

Technical Design describes :

* Hardware configuration
* Software needs
* Communication interfaces
* Input-Output of the system
* Software architecture
* Network architecture
* Any other thing that translates the requirements into a solution to the customer's problem

The design needs to be :

1. Correct and Complete
2. Understandable
3. At the right level
4. Maintainable

The following schema shows the transformation of INFORMAL Design to a detailed Design :

I. Informal design outline

II. Informal design

III. More formal design

IV. Finished design (final output)

HomeWork :

OBJECTIVES OF DESIGN

To negotiate system requirements
to set expectations with customers, marketing, and management personnel
Act as a blueprint during the development process
Guide the implementation tasks, including detailed design, coding, integration, and testing.

Why is Design Important?

The software design connects the program and user together.
It tells the program what the user wants, and tells the user what the program wants.
If the design is bad, the user will probably be disappointed and not be able to use the software. A good design is simple, intuitive and good-looking.

**Problem Partitioning**

(divide and conquer)

I. Modularity :
the optimal way of partitioning

A. A Modular system consists of well defined, manageable units, with well defined interfaces among the units

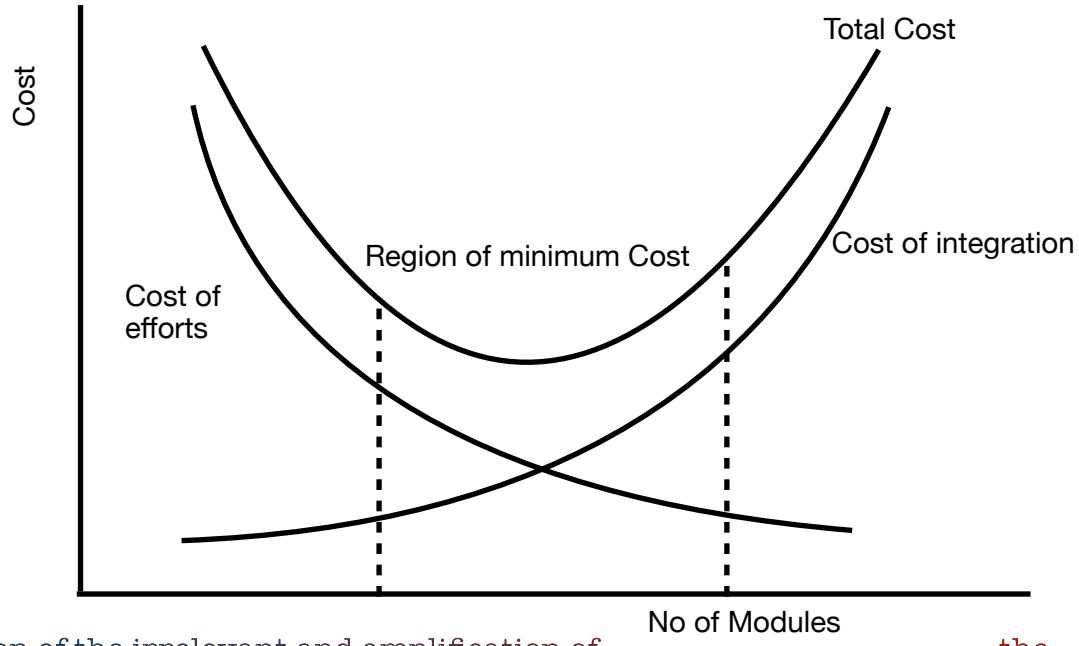
B. Modularity is the single attribute of software that allows a programme to be intellectually manageable

C. It **enhances Design Clarity**, which in turn eases implementation, debugging, testing, documentation and maintenance of software product

D. Simply dividing or chopping the software system into modules indefinitely won't make things simpler
This idea gives rise to the concept of modularity

E. Under modularity and over modularity should be avoided

II. Abstraction




Elimination of the irrelevant and amplification of the essential

Strategies for Design

1) Top-down design
start from the final output and go to the details

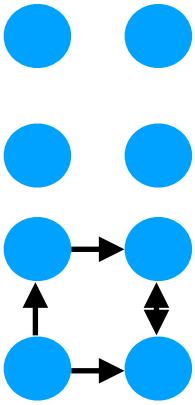
2) Bottom-up design
start from the details and build the final output

3) Hybrid design
both the above concepts

Modularity

Coupling | Cohesion

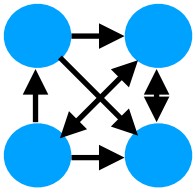
Module coupling : measure of the degree of interdependence between modules



Uncoupled modules

(no dependencies)

Loosely coupled



highly coupled

A good design has low coupling

Module Cohesion : how different units behave within (inside) the module

05/02/2020

III. Separation of concerns

Design concept that suggests that any complex problem can be more easily handled if it is subdivide into pieces that can each be solved and/or optimised independently.

A concern is a feature or behaviour that is specified as a part of the requirements model for the software.
By separating concerns into smaller (and therefore more manageable) pieces, a problem takes lesser time and effort to solve.

**Modularity** is the most common manifestation of separation of concerns.

Software is divided into separately named addressable components, sometimes called Modules, that are integrated to satisfy problem requirements.
Allows the programme to be intellectually manageable.

A System is considered **modular** if it consists of discrete components so that each component can be implemented separately and a change in one component has minimal impact on other components.

SRS —> deals with the problem | SDD —> deals with the solution

To produce Modular design, we must use some criteria to select modules.

Coupling and cohesion are two modularisation criteria

**Coupling** - how closely connected two routines or modules are; the strength of relationship between modules.

Measure of degree of interdependence between modules

Two modules with high coupling are strongly interconnected and thus, dependent on each other.

Loosely-coupled systems are made up of modules that are relatively independent. Highly-coupled systems share a great deal of dependence between modules. Uncoupled-modules have no interconnection at all.

Coupling is measured by the number of interconnections between modules.

e.g. coupling increases as number of calls between modules increases, or the amount of shared data increases.

**Cohesion** - level of strength and unity with which different components of a software program are inter-related. ( within the model )

A good design has **Low** coupling and **High** cohesion.

## different types of coupling and cohesion

19/02/2020

Mid-sem $\Longrightarrow$ first three units

Unit I - introductory ; SDLC ; software vs hardware

SDLC - 5 models

typical phases - requirements specs and analysis

design

implementation and unit testing

integration and system testing

operations and maintenance

Unit II - Requirements engineering — SRS

Unit III - Design — SDD

★ SRS vs SDD

★ Partitioning of problem / separation of concerns — affects cost and time

SDD in two levels {conceptual(for users/clients) and technical(for developers)}

may be clubbed if the user is technically proficient enough

modularity

**Types of Cohesion** ( ordered )

1. Functional cohesion (best)
2. Sequential cohesion
3. Communicational cohesion
4. Procedural cohesion
5. Temporal cohesion
6. Logical cohesion
7. Coincidental cohesion (worst)

Given a procedure that carries out operations X and Y, we define various forms of cohesion between X and Y.

**Procedural Cohesion**

X and Y are **structured** in the **same** way. This is a poor reason for putting them in the same procedure.(they have same structure but are not the same element)

Thus procedural cohesion occurs in modules whose instructions, although accomplish different tasks, yet have been combined because there is a specific order in which the tasks are to be completed.

e.g. if a report module includes (calculate GPA) and (print) — different tasks but put together

**Temporal Cohesion**

X and Y both must perform around the **same time**. A module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time span.

**Logical Cohesion**

X and Y perform **logically similar** operations. Therefore logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

**Coincidental Cohesion**

X and Y have no conceptual relation other than **shared code**. Coincidental Cohesion exists in modules that contain instructions that have little or no relationship to one another.

Good Design / Ideal Modularity — Low Coupling and High Cohesion

**Relationship between Coupling and Cohesion**

The essence of the design process is that the system is decomposed into parts to facilitate the capability of understanding and modifying a system.

Projects get into trouble because of massive requirement changes, these changes can be properly recognised and revealed.

If the software is not properly modularised, a host of seemingly trivial enhancement or changes will result into death of the project.

Therefore a good software design professes clean decomposition of a problem into modules, and the arrangement of these modules in a neat hierarchy.

Therefore a software engineer must design modules with a goal of high cohesion and low coupling.

e.g. plug and play capabilities in motherboard

## Strategy of Design

I. Bottom-up design strategy

   A. Build smaller components and gradually integrate to form the system

II. Top-down design strategy

   A. Start with the view of the system and gradually detail it with components

III. Hybrid Design

   A. Use the above approaches however suitable

A GOOD design

1. Easy to develop
2. Easy to modify later

20/02/2020

A system consists of components which have components of their own
a system is indeed a hierarchy of components.

The highest level component corresponds to the total system.

There are two possible approaches to design such a hierarchy:

**I)** Top down

**II)** Bottom up

The **top-down** approach starts from the highest level component of the hierarchy and proceeds through to lower levels.

A **bottom-up** approach starts with the lowest level component of the hierarchy and proceeds through higher levels progressively to the top level component.

Top-down approach

Starts by identifying major components of the system, decomposing them into their lower level components and iterating until the desired level of detail is achieved.

Top-down design methods often result in some form of step-wise refinement.

Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed, and the design can be implemented directly.

The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design.

Most design methodologies are based on the top-down approach.

Bottom-up approach

Starts with designing the most basic or primitive components and proceeds to higher level components that use these low level components.

Bottom-up methods work with layers of abstraction.

Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of these layers are then used to implement more powerful operations and a still higher level of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A Top-down approach is suitable only if the specifications of the system are clearly known and the system development if from scratch. However, if a system is to be built on an existing system, a Bottom-up approach is more suitable as it starts from some existing components.

**Hybrid design strategy**

Pure top-down or pure bottom-up approaches are often not practical, so for a bottom-up approach to be successful, we must have a good notion (idea) of the top to which the design should be headed. Without a good idea about the operations needed at the high layers, it is difficult to determine what operations the current layer should support.

For top-down approach to be effective, some bottom-up (mostly in the lowest level design layers) approach is essential for the following reasons :

i) To permit some common sub-modules

ii) Reuse of modules

Hybrid approach has really become popular after the acceptance of reusability of modules. Standard libraries, Microsoft foundation classes, object oriented concepts are the steps in this direction.

For next (24/02/2020) class :

- Dataflow diagrams
- Structure charts
- Structure approach
- Functional vs Object Oriented Approach of design