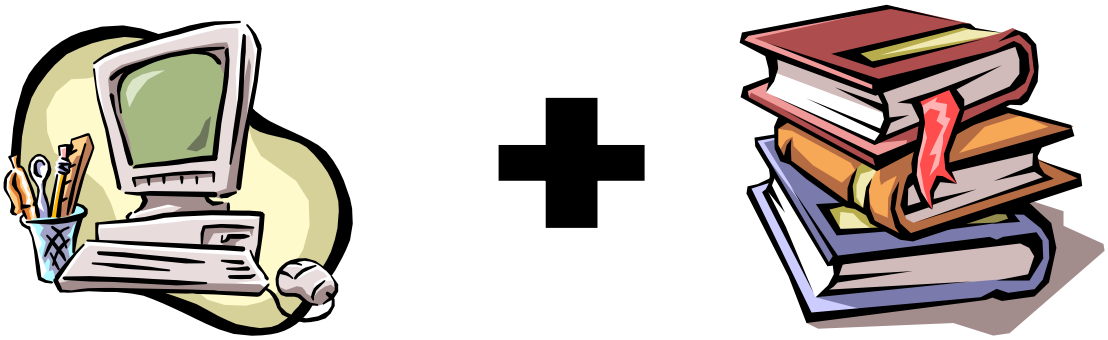
# INTRODUCTION TO SOFTWARE ENGINEERING

## WHAT IS SOFTWARE?

Computer programs and associated documentation



Software products may be developed for a particular customer or may be developed for a general market

Software products may be

- Generic - developed to be sold to a range of different customers
- Bespoke (custom) - developed for a single customer according to their specification

### What is software engineering?

The term ***software engineering*** is composed of two words, software and engineering.

**Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

**Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define **software engineering as** a detailed study of engineering to the design, development and maintenance of software. Software engineering was introduced to address the issues of low-quality software projects. Problems arise when a software generally exceeds timelines, budgets, and reduced levels of quality. It ensures that the application is built consistently, correctly, on time and on budget and within requirements.

**IEEE defines software engineering** as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

**Software engineers should**

- Adopt a systematic and organised approach to their work
- Use appropriate tools and techniques depending on
  - The problem to be solved,
  - The development constraints and
  - The resources available

**Software Characteristics**: - To gain an understanding of SW, it is important to examine the characteristics of software that make it different from other things that human being built. When hardware is built, the human creative Process (analysis, design, construction, testing) is ultimately translated into a physical form. Software is a logical rather than a physical system element. Therefore, software has characteristics that differ considerably from those of hardware:

- **Operational:** -This tells how good a software works on operations like budget, usability, efficiency, correctness, functionality, dependability, security and safety.
- **Transitional**: - Transitional is important when an application is shifted from one platform to another. So, portability, reusability and adaptability come in this area.
- **Maintenance**: - This specifies how good a software works in the changing environment. Modularity, maintainability, flexibility and scalability come in maintenance part.
- **Software is developed or engineered**; Software is developed or engineered; it is not manufactured in the classical sense. Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different.
- **Software doesn't "wear out.":** The hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (Ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.
- **Most software is custom-built**, rather than being assembled from existing components.

## Software Applications

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession. Information determinacy refers to the predictability of the order and timing of information.

- system software
- application software
- engineering/scientific software
- embedded software
- product-line software
- Web Apps (Web applications)
- AI software
- Legacy Software

**System software:** System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

**Real-time software**. Software that monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that

responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

**Engineering and scientific software**. Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

**Embedded software**. Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

**Web-based software**. The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

**Artificial intelligence software**. Artificial intelligence (AI) software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

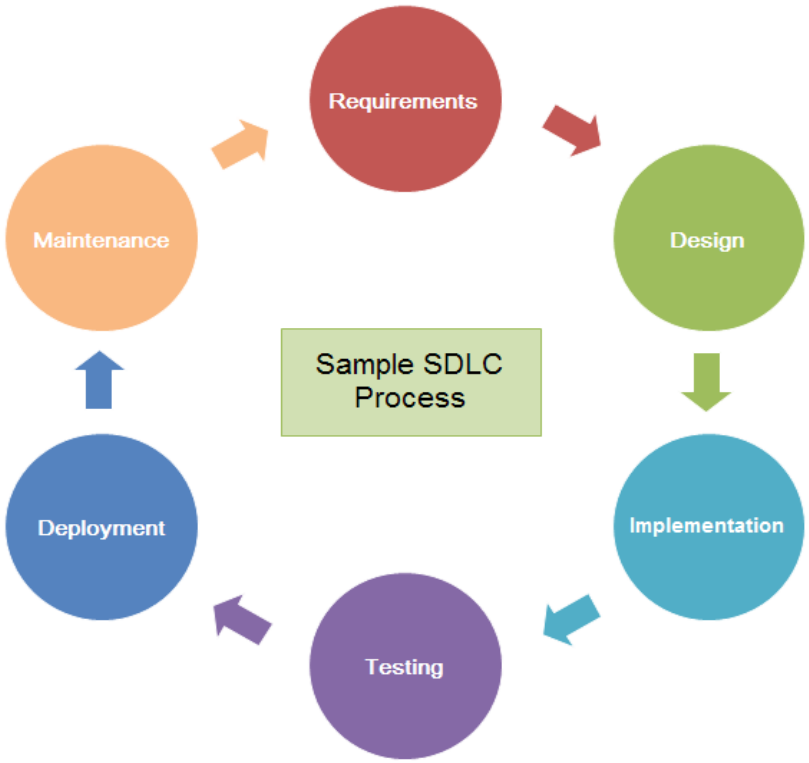
**Legacy Programs/ Legacy Software:** Today, a growing population of legacy programs is forcing many companies to pursue software reengineering strategies. The term legacy programs are a euphemism for older, often poorly designed and documented software that is

business critical and must be supported over many years. Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

## SDLC – Software Development Life Cycle:

The Software Development Lifecycle is a systematic process for building software that ensures the quality and correctness of the software built. SDLC process aims to produce high-quality software which meets customer expectations. The software development should be complete in the pre-defined time frame and cost. It consists of a detailed plan describing how to develop, maintain and replace specific software. Software life cycle models describe phases of the software cycle and the order in which those phases are executed. Each phase produces deliverables required by the next phase in the life cycle. A typical Software Development Life Cycle (SDLC) consists of the following phases:

1. Requirement gathering
2. System Analysis
3. Design
4. Development /Implementation or coding
5. Testing
6. Deployment
7. Maintenance




## 1. Requirement gathering:

- Requirement gathering and analysis is the most important phase in software development lifecycle. Business Analyst collects the requirement from the Customer/Client as per the client's business needs and documents the requirements in the Business Requirement Specification.
- This phase is the main focus of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements like; who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system?

## 2. Analysis Phase:

- Once the requirement gathering and analysis is done the next step is to define and document the product requirements and get them approved by the customer. This is done through SRS (Software Requirement Specification) document.
- SRS consists of all the product requirements to be designed and developed during the project life cycle.

- Key people involved in this phase are Project Manager, Business Analysist and Senior members of the Team.
- The outcome of this phase is Software Requirement Specification.

### 3. Design Phase:

- In this third phase the system and software design is prepared from the requirement specifications which were studied in the first phase.
- System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture.
- There are two kinds of design documents developed in this phase:
- **High-Level Design (HLD):** It gives the architecture of the software product to be developed and is done by architects and senior developers. It gives brief description and name of each module. It also defines interface relationship and dependencies between modules, database tables identified along with their key elements
- **Low-Level Design (LLD):** It is done by senior developers. It describes how each and every feature in the product should work and how every component should work. Here, only the design will be there and not the code. It defines the functional logic of the modules, database tables design with size and type, complete detail of the interface. Addresses all types of dependency issues and listing of error messages.

### 4. Coding/Implementation Phase:

- In this phase, developers start build the entire system by writing code using the chosen programming language.
- Here, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software Development Life Cycle process.
- In this phase, Developer needs to follow certain predefined coding guidelines. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.
- The outcome from this phase is Source Code Document (SCD) and the developed product.

### 5. Testing Phase:

- After the code is developed it is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase.
- They either test the software manually or using automated testing tools depends on process defined in STLC (Software Testing Life Cycle) and ensure that each and every component of the software works fine. The development team fixes the bug and send back to QA for a re-test. This process continues until the software is bug-free, stable, and working according to the business needs of that system.

### 6. Deployment:

After successful testing the product is delivered / deployed to the customer for their use. As soon as the product is given to the customers they will first do the beta testing. If any changes are required or if any bugs are caught, then they will report it to the engineering team. Once those changes are made or the bugs are fixed then the final deployment will happen.

### 7. Maintenance:

Software maintenance is a vast activity which includes optimization, error correction, and deletion of discarded features and enhancement of existing features. Since these changes are necessary, a mechanism must be created for estimation, controlling and making modifications. The essential part of software maintenance requires preparation of an accurate plan during the development cycle. Typically, maintenance takes up about 40-80% of the project cost, usually closer to the higher pole. Hence, a focus on maintenance definitely helps keep costs down.

## Software Process:

A structured set of activities required to develop a software system. There are many different software processes but all involve:

- Specification – defining what the system should do;
- Design and implementation – defining the organization of the system and implementing the system;
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

## Software Process Model:

A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective. When we describe and discuss processes, we usually think about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities. There are different models available.

### Water fall Model (Linear-Sequential Life Cycle Model):

- The Waterfall Model was first Process Model to be introduced. It is very simple to understand and use. In a Waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.
- Waterfall model is the earliest SDLC approach that was used for software development.
- In "The Waterfall" approach, the whole process of software development is divided into separate phases.
- The outcome of one phase acts as the input for the next phase sequentially.
- This means that any phase in the development process begins only if the previous phase is complete.
- At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project.

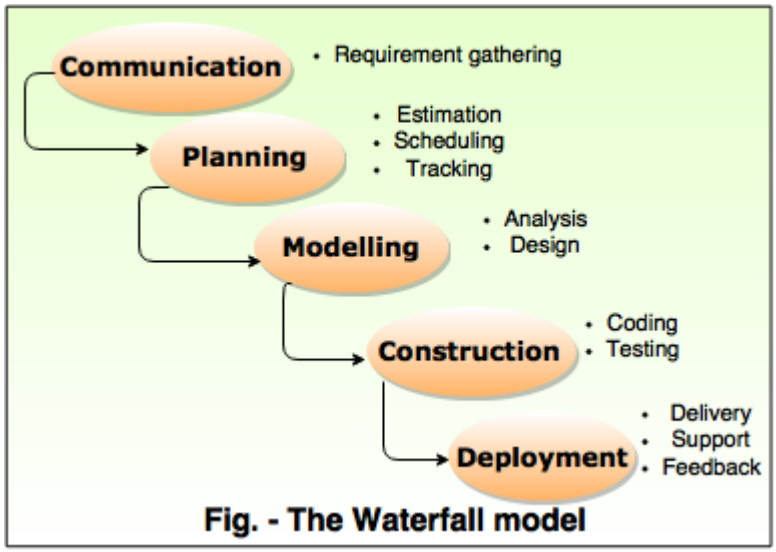



Fig. - The Waterfall model

**1. Communication:** The major task performed is requirement gathering which helps in finding out the exact need of the customer. Once all the needs of the customer are gathered the next step is planning.

**2. Planning:** Major activities like planning for schedule, keeping tracks on the processes and the estimation related to the project are done. Planning is even used to find the types of risks involved throughout the projects. Planning describes how technical tasks are going to take place and what resources are needed and how to use them.

**3. Modeling:** This is one the important phases of the architecture of the system is designed in this phase. Analysis is carried out and depending on the analysis a software model is designed. Different models for developing software are created depending on the requirements gathered in the first phase and the planning done in the second phase.

**4. Construction:** The actual coding of the software is done in this phase. This coding is done on the basis of the model designed in the modeling phase. So in this phase software is actually developed and tested.

**5. Deployment:** In this last phase the product is actually rolled out or delivered & installed at customer's end and support is given if required. A feedback is taken from the customer to ensure the quality of the product.

**Advantages of waterfall model:**

- This model is simple and easy to understand and use.
- It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- In this model phases are processed and completed one at a time. Phases do not overlap.
- Waterfall model works well for smaller projects where requirements are very well understood.

**Disadvantages of waterfall model:**

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

When to use the waterfall model

- This model is used only when the requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

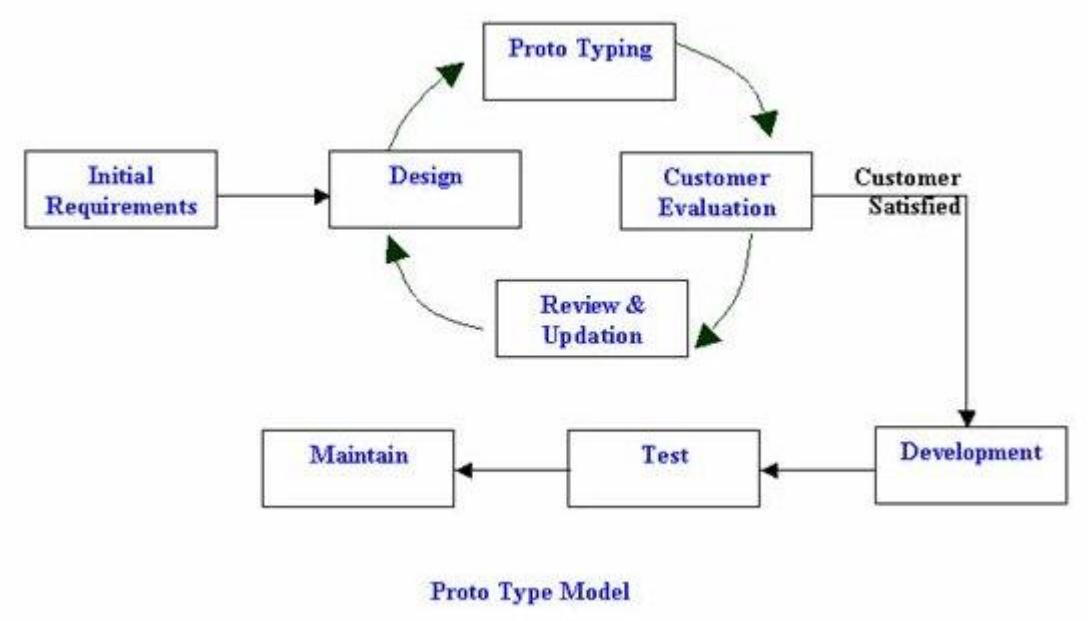
## Prototyping Model:

**Prototype**: Prototype is an early approximation of a final system/product or Prototype is a working model of software with some limited functionality.

**The Software Prototyping** refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

1. The basic idea in **Prototype model** is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements.
2. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.
3. The users thoroughly evaluate the first prototype, noting its strengths and weaknesses, what needs to be added, and what should to be removed. The developer collects and analyzes the remarks from the users.
4. The first prototype is modified, based on the comments supplied by the users, and a second prototype of the new system is constructed.
5. The second prototype is evaluated in the same manner as was the first prototype.
6. The process of refining the prototype is repeated till all the requirements of users are met.

7. When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype and it is thoroughly evaluated and tested.




Proto Type Model

**Advantages of Prototyping Model**

- ✓ In the development process of this model users are actively involved.
- ✓ The development process is the best platform to understand the system by the user.
- ✓ Earlier error detection takes place in this model.
- ✓ It gives quick user feedback for better solutions.
- ✓ It identifies the missing functionality easily. It also identifies the confusing or difficult functions.

**Disadvantages of Prototyping Model**

- ✓ The client involvement is more and it is not always considered by the developer.
- ✓ It is a slow process because it takes more time for development.
- ✓ Many changes can disturb the rhythm of the development team.
- ✓ It is a throw away prototype when the users are confused with it.

**When to use Prototype model:**

- ✓ Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- ✓ Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- ✓ Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

## RAD model:

- ➢ RAD model is ***Rapid Application Development*** model. It is a type of incremental model.
- ➢ In RAD model the components or functions are developed in parallel as if they were mini projects
- ➢ The developments are time boxed, delivered and then assembled into a working prototype.
- ➢ This can quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements.

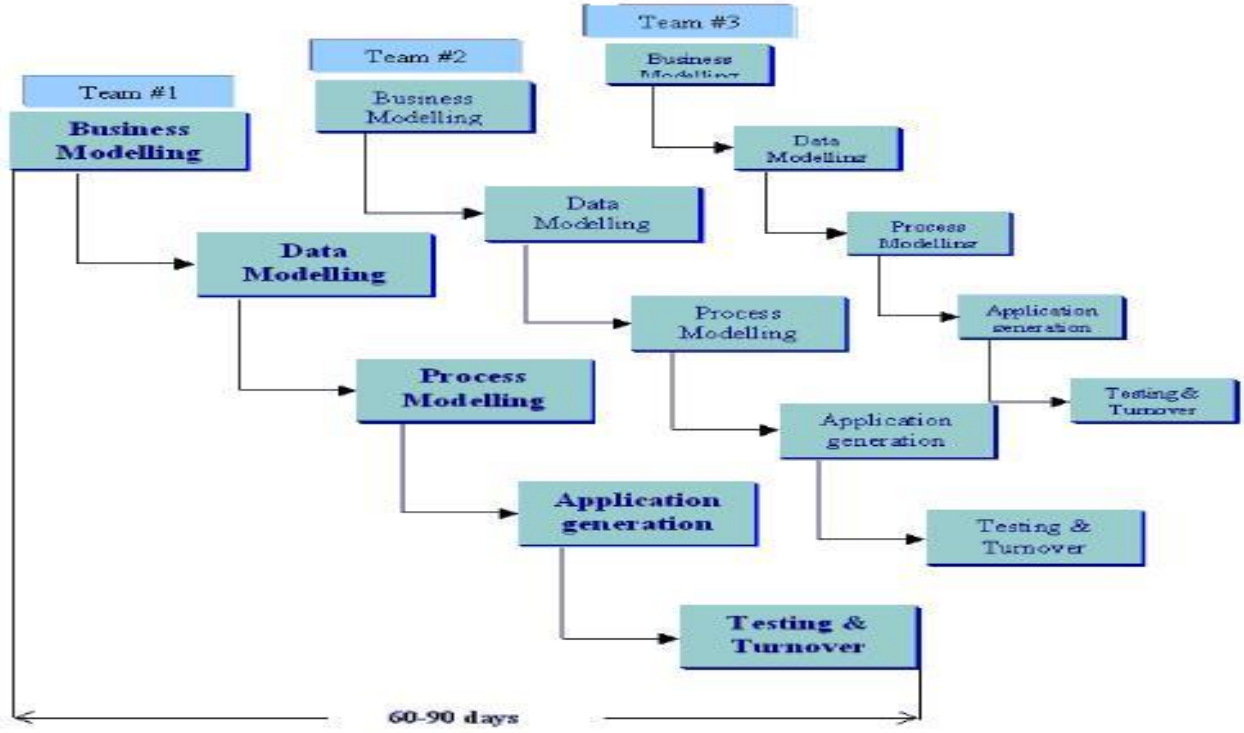



Figure 1.5 – RAD Model

The phases in the rapid application development (RAD) model are:

- **Business modeling:** The information flow is identified between various business functions.
- **Data modeling:** Information gathered from business modeling is used to define data objects that are needed for the business.
- **Process modeling:** Data objects defined in data modeling are converted to achieve the business information flow to achieve some specific business objective. Descriptions are identified and created for CRUD of data objects.
- **Application generation:** Automated tools are used to convert process models into code and the actual system.
- **Testing and turnover:** Test new components and all the interfaces.

**Advantages of the RAD model:**

- Reduced development time.
- Increases reusability of components
- Quick initial reviews occur

- Encourages customer feedback
- Integration from very beginning solves a lot of integration issues.

**Disadvantages of RAD model:**

- Depends on strong team and individual performances for identifying business requirements.
- Only system that can be modularized can be built using RAD
- Requires highly skilled developers/designers.
- High dependency on modeling skills
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.

**When to use RAD model**

- RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

## Incremental Process model:

- The incremental model combines the elements of waterfall model and they are applied in an iterative fashion.
- The first increment in this model is generally a core product.
- Multiple development cycles take place. Cycles are divided up into smaller, more easily managed modules. Each module passes through the requirements, design, implementation and testing phases.
- Each increment builds the product and submits it to the customer for any suggested modifications.
- The next increment implements on the customer's suggestions and add additional requirements in the previous increment.

- This process is repeated until the product is finished. For example, the word-processing software is developed using the incremental model.

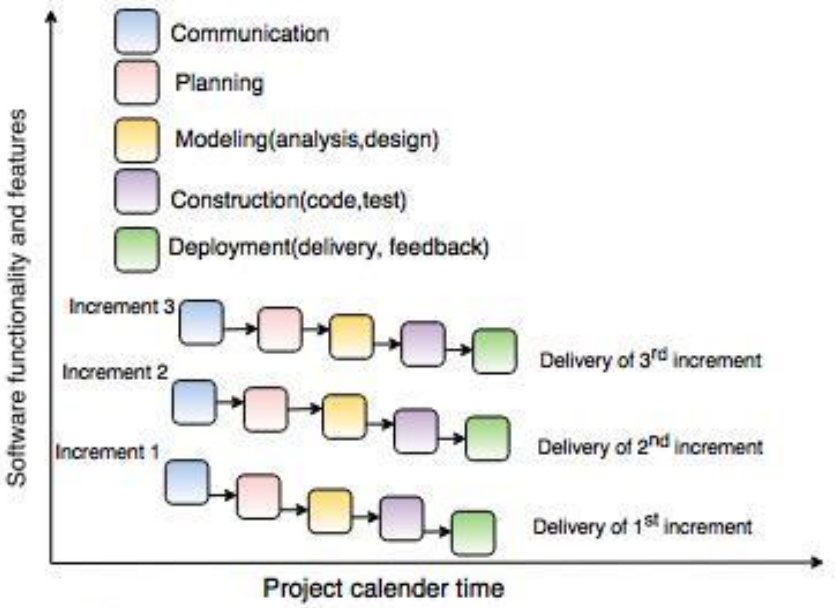



**Fig. - Incremental Process Model**

**Advantages of incremental model**

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug during the smaller iteration.
- The working software generates quickly and early during the software life cycle.
- The customers can respond to its functionalities after every increment.

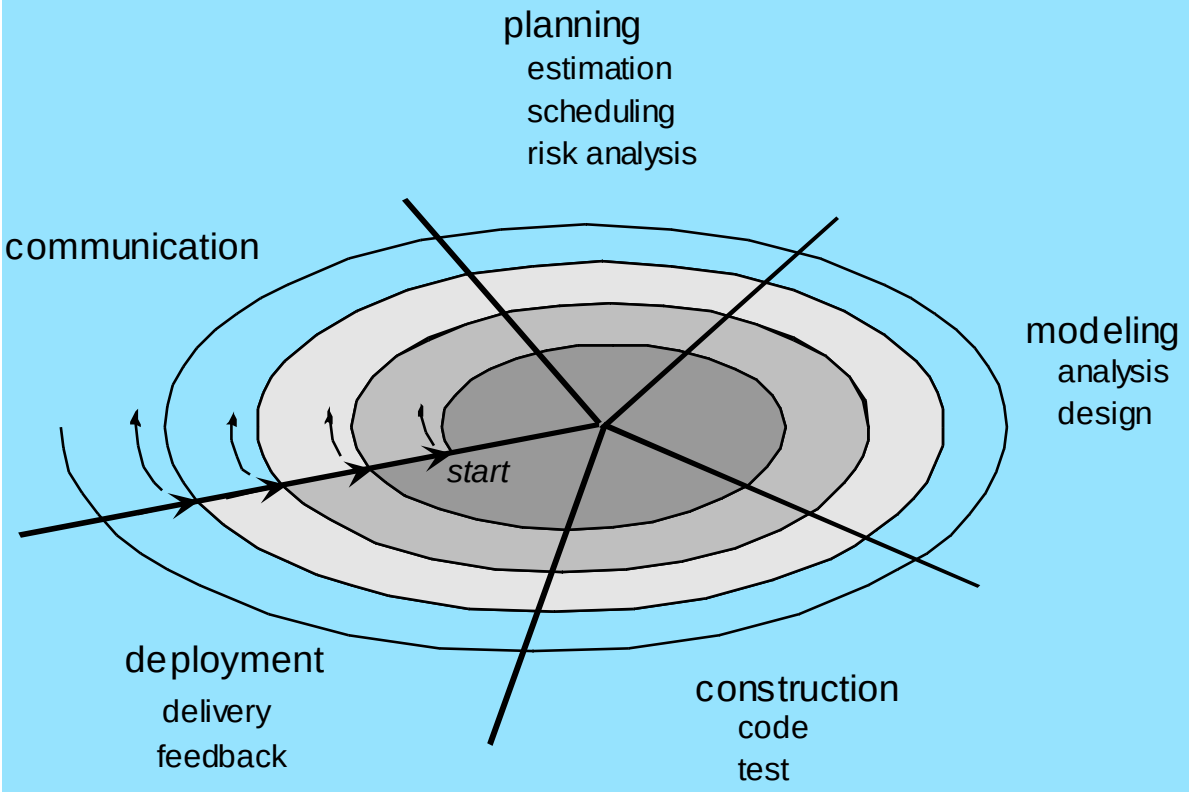
**Disadvantages of the incremental model**

- The cost of the final product may cross the cost estimated initially.
- This model requires a very clear and complete planning.
- The planning of design is required before the whole system is broken into small increments.
- The demands of customer for the additional functionalities after every increment causes problem during the system architecture.

## Spiral Model:

- The spiral model was first mentioned by Barry Boehm in his 1986 paper.
- The spiral model is similar to the incremental model, with more emphasis placed on risk analysis.
- The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation.

- A software project repeatedly passes through these phases in iterations (called Spirals in this model).
- The baseline spiral, starting in the planning phase, requirements is gathered and risk is assessed. Each subsequent spiral builds on the baseline spiral.
- Each phase in spiral model begins with a design goal and ends with the client reviewing the progress.




Spiral Model Phases:

| Spiral Model Phases | Activities performed during phase |
|---|---|
| Planning | ➢ It includes estimating the cost, schedule and resources for the iteration.<br>➢ It also involves understanding the system requirements for continuous communication between the system analyst and the customer<br>➢ Requirements are gathered from the customers and the objectives are identified, elaborated and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant. |
| Risk Analysis | ➢ Identification of potential risk is done while risk mitigation strategy is planned and finalized |

| | |
|---|---|
| | ➢ During the second quadrant all the possible solutions are evaluated to select the best possible solution.<br>➢ Then the risks associated with that solution is identified and the risks are resolved using the best possible strategy.<br>➢ At the end of this quadrant, Prototype is built for the best possible solution. |
| Engineering | ➢ It includes testing, coding and deploying software at the customer site<br>➢ During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available. |
| Evaluation | ➢ Evaluation of software by the customer. Also, includes identifying and monitoring risks such as schedule slippage and cost overrun.<br>➢ In the fourth quadrant, the Customers evaluate the so far developed version of the software.<br>➢ In the end, planning for the next phase is started. |

## When to use Spiral Methodology?

- ✓ When project is large
- ✓ When releases are required to be frequent.
- ✓ When risk and costs evaluation is important
- ✓ For medium to high-risk projects
- ✓ When requirements are unclear and complex
- ✓ When changes may require at any time

## Advantages and Disadvantages of Spiral Model

| Advantages | Disadvantages |
|---|---|
| • Additional functionality or changes can be done at a later stage | • Risk of not meeting the schedule or budget |
| • Cost estimation becomes easy as the prototype building is done in small fragments | • It works best for large projects only also demands risk assessment expertise |
| • Continuous or repeated development helps in risk management | • For its smooth operation spiral model protocol needs to be followed strictly |
| • Development is fast and features are added in a systematic way | • Documentation is more as it has intermediate phases |

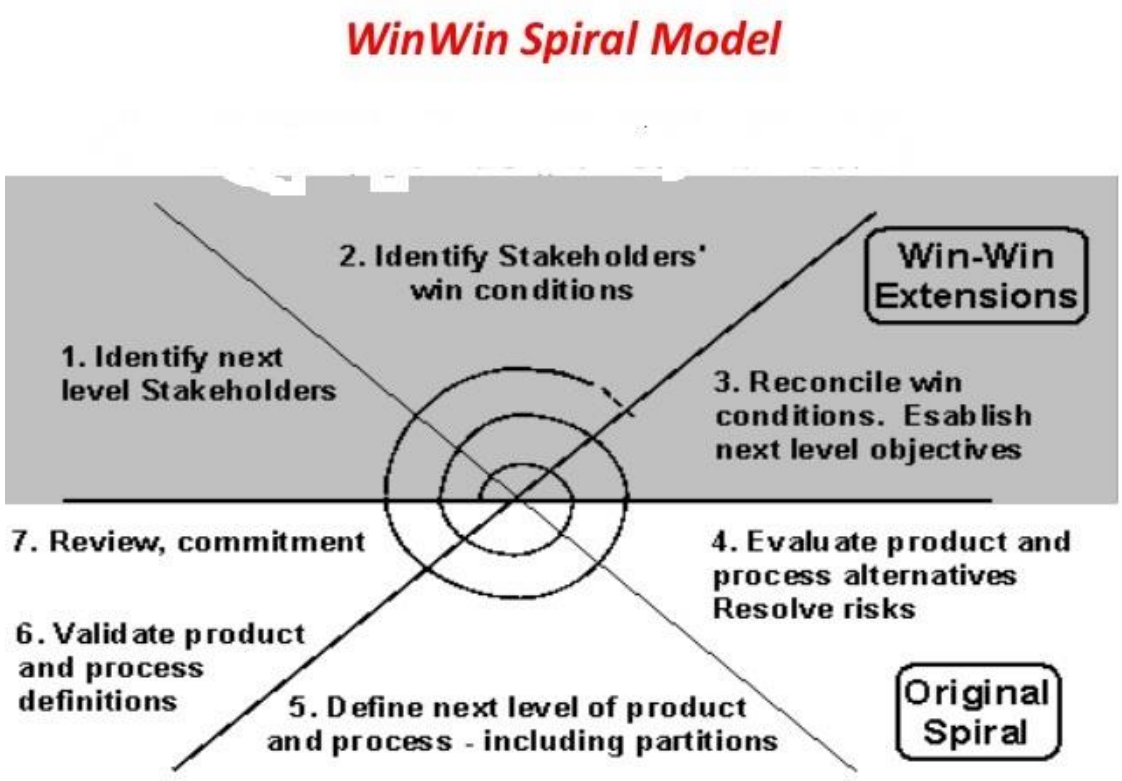
| | |
|---|---|
| • There is always a space for customer feedback | • It is not advisable for smaller project, it might cost them a lot |

## Win-Win Spiral Methodology:

- The win-win spiral approach is very similar to the spiral model in that it is simply an extension of the spiral model. In the win-win model approach, everyone discusses things together to figure out what should go into the new version of the software.
- The WinWin spiral software engineering methodology expands the **Boehm-Spiral methodology** by adding a priority setting step, the WinWin process, at the beginning of each spiral cycle and by introducing intermediate goals, called anchor points that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.
- The WinWin Spiral Model uses **Theory W (win-win)** to develop software and system requirements, and architectural solutions, as win conditions negotiated among a project's stakeholders (user, customer, developer, maintainer, interface, *etc.*).
- Boehm's WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following **activities are defined:**

  1. Identification of the system or subsystem's key "stakeholders."
  2. Determination of the stakeholders' "win conditions."
  3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team).
- Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.

- In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestone-called anchor points. T**he anchor points represent three different views of progress** as the project traverses the spiral.
- The first anchor point, **life cycle objectives (LCO),** defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements.
- The second anchor point, l**ife cycle architecture (LCA)**, establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions.
- **Initial operational capability (IOC)** is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.



Win win situation can let you get short term benefit but it has its own drawbacks. Some of these listed below.

**Advantages:**

1 high amount of risk analysis hence,

avoidance of risk is enhanced

2 good for large and mission critical projects

3 strong approval and documentation control

**Disadvantages:**

1 can be a costly model to use

2 risk analysis requires highly specific expertise

3 projects success is highly depends on the risk analysis phase

Ex: Two people were asked to accomplish a job. If one is giving his best and other is just doing it, then implementing win win for their performance review will give an impact that working hard doesnt matter. It will create a mental delema that if working hard and working "Just OK" results in same, then there is no point in working hard. working ok will do. it leads to Loyalty loss. when a perticular employee's extra efforts are not being treated in proper manner.

## Unit – II

**Project Management Concepts**: The Management Spectrum: People, Process, Project and Product

**Software Project Planning**: Project planning objectives, Software Scope, Resources, Software Project Estimation, The Make-Buy Decision, Software Risks.

**Software Project Management:**

- Software Project management involves planning, monitoring and control of people, process and event that occurs as software evolves from preliminary concept to fully operational deployment.
- The Software Project management Spectrum focuses on the four P's:
  1. People
  2. Product
  3. Process
  4. Project

**Refer PPT (forwarded) for notes on 4 Ps**

**Reasons for the failure of software project**

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change.
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost.
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

## Software Project Planning

Project planning is the process of identifying and planning the activities required to build a software product. Planning will improve a project's outcome, it requires you to make an initial commitment. Effective planning is needed to resolve problems upstream [early in the project] at low cost, rather than downstream [late in the project] at high cost. Planning is one of the most important management activities and is an ongoing effort throughout the life of the project.

**Project Planning Objectives**

- The objective of software project planning is to provide a framework that enables the Manager to make reasonable estimates of:
    - Estimation of resources
    - Estimation of cost
    - Estimation of schedule

These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses.

- In addition, estimates should attempt to define best case and worst-case scenarios so that project outcomes can be bounded.
- The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking and monitoring a complex technical project.
- The purpose of the project planning is to ensure that the end result is completed on time, within budget and exhibits quality.

**Task Set for Project Planning**

1. Establish Project Scope.
2. Determine Feasibility.
3. Analyze risks
4. Define required resources.
    a. Determine required human resources.
    b. Define reusable software resources.
    c. Identify environmental resources.
5. Estimate cost and effort.
    a. Decompose the problem.
    b. Develop two or more estimates using size, function points, process tasks, or use cases.
    c. Reconcile the estimates.
6. Develop a project schedule
    a. Establish a meaningful task set.
    b. Define a task network.
    c. Use scheduling tools to develop a time-line chart.
    d. Define schedule tracking mechanisms.

**SOFTWARE SCOPE**

- Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.
- Software scope describes
    - The functions and features that are to be delivered to end users
    - The data that are input and output
    - The “content” that is presented to users as a consequence of using the software;

- The performance, constraints, interfaces, and reliability that bound the system.

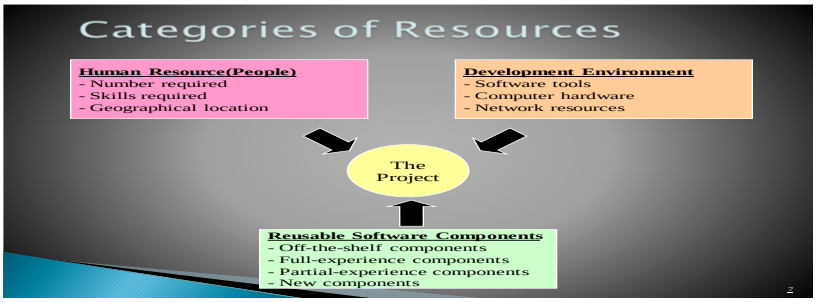
- Scope is defined using one of two techniques:
  1. A narrative description of software scope is developed after communication with all stakeholders.
  2. A set of use cases is developed by end users.
- Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation.
- Performance considerations encompass processing and response time requirements.
- Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.
- Scope is defined by answering the following questions:
  **Context**. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?
  **Information objectives.** What customer-visible data objects are produced as output from the software? What data objects are required for input?
  **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?
- Software project scope must be unambiguous and understandable at the management and technical levels.
- A statement of software scope must be bounded.

**SOFTWARE FEASIBILITY**

- Once scope has been identified project feasible analysis need to be made.
- Software feasibility has four dimensions
  - **Technology** –Technical feasibility concentrates on the availability of technology (software and hardware) and cost of technology.
  - **Finance** – Financially feasibility concentrates on the cost incurred for the development of software product. cost benefit analysis is made to make a decision.
  - **Time** –The time taken to build a software product is estimated and it is verified whether l the project's time-to-market beat the competition?
  - **Resources** – Does the software organization have the resources needed to succeed in doing the project?

**DEFINE REQUIRED RESOURCES.**

- The important planning task is estimation of the resources required to accomplish the software development effort.
- Each resource is specified with
    - A description of the resource
    - A statement of availability
    - The time when the resource will be required
    - The duration of time that the resource will be applied
  - Three major categories of software engineering resources
    a. Determine required human resources.
    b. Define reusable software resources.
    c. Identify environmental resources.




**Human Resources**

- The Planners need to select the number and the kind of people skills needed to complete the project
- They need to specify the organizational position and job specialty for each person
- Small projects of a few person-months may only need one individual
- Large projects spanning many person-months or years require the location of the person to be specified also
- The number of people required can be determined only after an estimate of the development effort

**Reusable Software Resources**

- Component-based software engineering emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.
- Reusable software components can be classified into the following categories
  **Off-the-shelf components**
    - Components are from a third party or were developed for a previous project
    - Ready to use; fully validated and documented; virtually no risk

**Full-experience components**

- Components are similar to the software that needs to be built
- Software team has full experience in the application area of these components
- Modification of components will incur relatively low risk

**Partial-experience components**

- Components are related somehow to the software that needs to be built but will require substantial modification
- Software team has only limited experience in the application area of these components
- Modifications that are required have a fair degree of risk

**New components**

- Components must be built from scratch by the software team specifically for the needs of the current project
- Software team has no practical experience in the application area
- Software development of components has a high degree of risk

**Environmental Resources.**

- A software engineering environment (SEE) incorporates hardware, software, and network resources that provide platforms and tools to develop and test software work products.
- Most software organizations have many projects that require access to the SEE provided by the organization
- Planners must identify the time window required for hardware and software and verify that these resources will be available

**PROJECT ESTIMATION**

- Project Estimation is the process of estimating the cost , time, resources and effort required to build a software product.
- The accuracy of a software project estimate is predicated on
  - The degree to which the planner has properly estimated the size (e.g., KLOC) of the product to be built
  - The ability to translate the size estimate into human effort, calendar time, and money
  - The degree to which the project plan reflects the abilities of the software team
  - The stability of both the product requirements and the environment that supports the software engineering effort
- Project Estimation is done through the following steps

  a. Decompose the problem.

  b. Develop two or more estimates using size, function points, process tasks, or use cases.

c. Reconcile the estimates.

- Project Estimation can be performed in the following ways
  - Delay estimation until late in the project (we should be able to achieve 100% accurate estimates after the project is complete)
  - Base estimates on similar projects that have already been completed
  - Use relatively simple decomposition techniques to generate project cost and effort estimates
  - Use one or more empirical estimation models for software cost and effort estimation.

## Explain various categories of SOFTWARE RISKS

- Risk Analysis and management are activities that help a software team to understand uncertainty.
- Risk exhibits two characteristics
  **1. Uncertainty**: The occurrence of risk is uncertain that is risk may or may not occur.
  **2. Loss**: If the risk becomes a reality, unwanted consequences or losses will occur.
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk.

**Different categories of risks are:**

**Project risks:**

- They threaten the project plan.
- That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase.
- Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

**Technical risks:**

✓ They threaten the quality and timeliness of the software to be produced.
✓ If a technical risk becomes a reality, implementation may become difficult or impossible.
✓ Technical risks identify potential design, implementation, interface, verification, and Maintenance problems.

**Business risks :**

✓ They threaten the viability of the software to be built and often jeopardize the project or the Product.

Candidates for the top five business risks are

(1) **Market risk:** building an excellent product or system that no one really wants

(2) **Strategic risk:** building a product that no longer fits into the overall business strategy for the company

(3) **Sales risk:** building a product that the sales force doesn't understand how to sell.

(4) **Management risk:** losing the support of senior management due to a change in focus or a Change in people

(5) **Budget risks:** losing budgetary or personnel commitment.

- ✓ **Known risks** are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

- ✓ **Predictable risks** are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

- ✓ **Unpredictable risks** are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

# UNIT - III

Analysis Concepts Principles: Requirements Analysis, Requirements Elicitation for Software-Initiating the Process, Facilitated Application Specification Techniques, Quality Function Deployment, Use Cases, Analysis Principle's, The Software Requirement Specification (Ch 11)

## Software Analysis Concepts and Principles

**Software Analysis Concepts and Principles**

The overall role of software in large system is identified during system engineering. However, it's necessary to take a harder look at software's role to understand the specific requirements that must be achieved to build high-quality software. That's the job of software requirements analysis. To perform the job properly you should follow a set of underlying concepts and principles.

**1. Requirements Analysis**

- Requirement analysis is a software engineering task that **bridges the gap between** system level **requirements engineering** and **software design**.
- Requirements engineering activities result in the specification of software's operational characteristics, indicate software's interface with other system elements, and establish constraints that software must meet.
- Requirement analysis allows the software engineer to refine domains that will be treated by software.
- Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs.

- Software requirements analysis may be divided into five areas of effort:

(1) problem recognition,

(2) evaluation and synthesis,

(3) modeling,

(4) specification, and

(5) review.

- The analyst studies the *system specification* and the *software Project Plan*. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates.
- Problem evaluation and solution synthesis is the next major area of effort for analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints.
- Throughout evaluation and solution synthesis, the analyst's primary focus is on "what" not "how". What data does the system produce and consume, what functions must the system perform, what behavior does the system exhibit, what interfaces are defined and what constraints apply?
- During the evaluation and solution synthesis activity, the analyst creates models of the system in an effort to better understand data and control flow, functional processing, operational behavior, and information content.
- The model serves as a foundation for software design and as the basis for the creation of specifications for the software.

**2. Requirements Elicitation for Software**

- Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process.
- A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help.

- Communication Analysis techniques are as follows

**2.1 Initiating the Process**

- The most commonly used requirements elicitation technique is to conduct a meeting or interview.
- The first meeting between a software engineer (the analyst) and the customer can be awkward.
- The analyst starts by asking context-free questions. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.
- The first set of context-free questions focuses on the customer, the overall goals, and the benefits. For example, the analyst might ask:
- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?
- These questions help to identify all stakeholders who will have interest in the software to be built.
- In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.
- The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:
- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?
- The final set of questions focuses on the effectiveness of the meeting.
- These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful analysis.

- But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

**2.2 Facilitated Application Specification Techniques(FAST)**

- Customers and software engineers have an unconscious "us and them" mind-set. With these problems in the mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification called facilitated application technique (FAST).
- Facilitated application specification techniques (FAST), approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.
- FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.
- Basic guidelines for this technique are:
    - A meeting is conducted at a neutral site and attended by both software engineers and customers.
    - Rules for preparation and participation are established.
    - An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
    - A "facilitator" controls the meeting.
    - A "definition mechanism" is used
    - The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conductive to the accomplishment of the goal.

- A meeting place, time, and date for FAST are selected and a facilitator is chosen. Attendees from both the development and customer/user organizations are invited to attend. The product request is distributed to all attendees before the meeting date.
- Initial meeting between the developer and customer occur and **basic questions and answers** help to establish the scope of the problem and the overall perception of a solution. The product request distributed to all attendees before the meeting date.
- The FAST team is composed of representatives from marketing, software and hardware engineering, and manufacturing.
- As the FAST meeting begins, the first topic of discussion is the need and justification for the new product – everyone should agree that the product justified. Once agreement has been established, each participant his or her list for discussion.
- After individual lists are presented in one topic area, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion, but does not delete anything. The combined list is shortened, lengthened, or reworded to properly reflect the product or system to be developed.
- The objective is to develop a *consensus list* in each topic area. Each sub team presents its mini-specs to all FAST attendees for discussion. After the mini-specs are completed, each FAST attendee makes a list of *validation criteria* for the product or system and presents his or her to the team.

**2.3 Quality Function Deployment(QFD)**

- *Quality function deployment (*QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD "concentrates on maximizing customer satisfaction from the software engineering process.
- QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.
- QFD identifies three types of requirements:

1. **Normal requirements.** The **objectives and goals** that are stated for a product or system during meeting with customer. If these requirements are present, the customer is satisfied.

   **Examples** of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

2. **Expected requirements.** These requirements **are implicit** to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.

   **Examples** of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

3. **Exciting requirements.** These features go **beyond the customer's expectations** and prove to be very satisfying when present.

   **For example**, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

*Functional deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment* examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

**2.4. Use-Cases**

- As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed.
- The scenarios, often called use-cases, provide a description of how the system will be used.

- To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These actors actually represent roles that people (or devices) play as the system operates.
- Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. It's most important to note that an actor and a user are not the same thing.
- An actor represents a class of external entities that play just one role. Once actors have been identified, use-case can be developed.
- The use-case describes the manner in which an actor interacts with the system. The use-case should be answer below questions:
    - What main tasks or functions are performed by an actor?
    - What system information will the actor acquire, produce, or change?
    - Will the actor have to inform the system about changes in the external environment?
    - What information does the actor desire from the system?
    - Does the actor wish to be informed about unexpected changes?

In general, use-case is simply a written narrative that describes the role of an actor as interaction with the system occurs.

**3. Analysis Principles**

Over the past two decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view.

- The information domain of a problem must be represented and understood.
- The functions that the software is to perform must be defined.
- The behavior of the software must be represented.

- The models that depict information function and behavior must be partitioned in a manner that uncovers details in a layered fashion.
- The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that function may be understood more completely.

Models are used so that the characteristics of function and behavior can be communicated in a compact fashion. Partitioning is applied to reduce complexity. Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

In addition to these operational analysis principles for requirements engineering are:

- Understand the problem before you begin to create the analysis model.
- Develop prototype that enable a user to understand how human/machine interaction will occur.
- Record the origin of and the reason for every requirement. This is the first step in establishing traceability back to the customer.
- Use multiple views of requirements. Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.
- Rank requirements.
- Work to eliminate ambiguity. . Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity.

## 4.0 Specification

### Specification principles

- Separate functionality from implementation
- Develop a model of the desired behavior of a system that encompasses data and functional response of a system to various stimuli from the environment
- Establish the context in which software operates by specifying the manner in which other systems components interact with software
- Define the environment in which the system operates and indicate how “a highly intertwined collection of agents react to stimuli in the environment (changes to objects) produced by those agents”
- Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community
- Recognize that “the specifications must be tolerant of incompleteness and augmentable.” A specification is always a model-an abstraction-of some real situation that is normally quite complex. Hence, it will be incomplete and will exist at many level of detail
- Establish the content and structure of a specification in a way that will enable it to be amenable to change

### Representation

- Representation format and content should be relevant to the problem
- Information contained within the specification should be nested
- Diagrams and other notational forms should be restricted in number and consistent in use
- Representations should be revisable

### The software requirements specification

- Is produced at the culmination of analysis task.
- The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.
- Format of software requirements specification:
    - Introduction
    - Information description
    - Functional description
    - Behavioral description
    - Validation criteria
    - Bibliography and appendix

## 5.0 Specification Review

- A review of the Software Requirements Specification is conducted by both the software developer and the customer.
- Because the specification forms the foundation of the development phase, extreme care should be taken into conducting the review.
- Once the review is complete, the Software Requirements Specification is "signed-off" by both the customer and developer.

The specification becomes a "contract" for software development

# UNIT IV

**Design Concepts Principles:** The Design Process, Design Principles, Design Concepts – Abstraction, Refinement, Modularity, Software Architecture, Control Hierarchy, Structural Partitioning, Data Structure, Software Procedure, Information Hiding, Effective Modular Design – Functional Independence, Cohesion, Coupling (Ch 13)

## Design Process:

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.
- The main aim of design engineering is to generate a model which shows firmness, delight and commodity.
- Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system.
- The design process comprises a set of principles, concepts and practices, which allow a software engineer to model the system or product that is to be built. This model, known as design model, is assessed for quality and reviewed before code is generated and tests are conducted.
- **IEEE** defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'

## Three characteristics serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

**Quality Guidelines**

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

1. A design should exhibit an architecture that:
    - Has been created using recognizable architectural styles or patterns,
    - Is composed of components that exhibit good design characteristics
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be represented using a notation that effectively communicates its meaning.

Design Principles:

Some of the commonly followed design principles are as following.

1. **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
2. **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.

3. **Software design should be uniform and integrated:** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.
4. **Software design should be flexible:** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
5. **Software design should ensure minimal conceptual (semantic) errors:** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
6. **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
7. **Software design should represent correspondence between the software and real-world problem:** The software design should be structured in such away that it always relates with the real-world problem.
8. **Software reuse:** Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
9. **Designing for testability:** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.

10. **Prototyping:** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

## Software Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods represent the manner through which the concepts are applied. As new technology replaces older technology, many changes occur in the methods that are used to apply the concepts for the development of software. However, the fundamental concepts underlining the software design process remain the same, some of which are described here.

## Abstraction

- Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components.
- **IEEE** defines abstraction as a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.
- The concept of abstraction can be used in two ways: as a process and as an entity. As a **process,** it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an **entity,** it refers to a model or view of an item.

- Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail.
- For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction. All these mechanisms allow us to control the complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

1. **Functional abstraction:** This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.
2. **Data abstraction:** This involves specifying data that describes a data object. For example, the data object *window* encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.
3. **Control abstraction:** This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

## Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

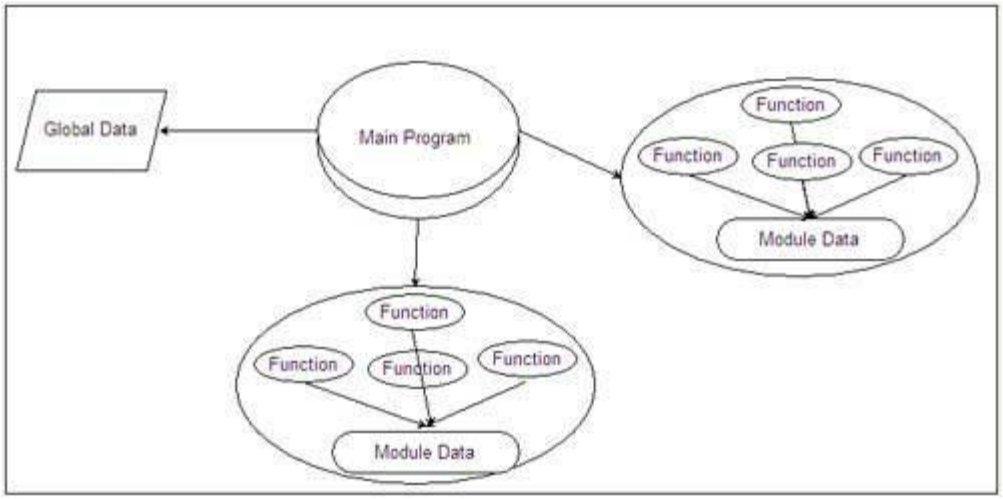
- Provides an insight to all the interested stakeholders that enable them to communicate with each other
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure (for supporting particular architectural styles) and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario.

Note that software architecture comprises two elements of design model, namely, data design and architectural design.

## Modularity

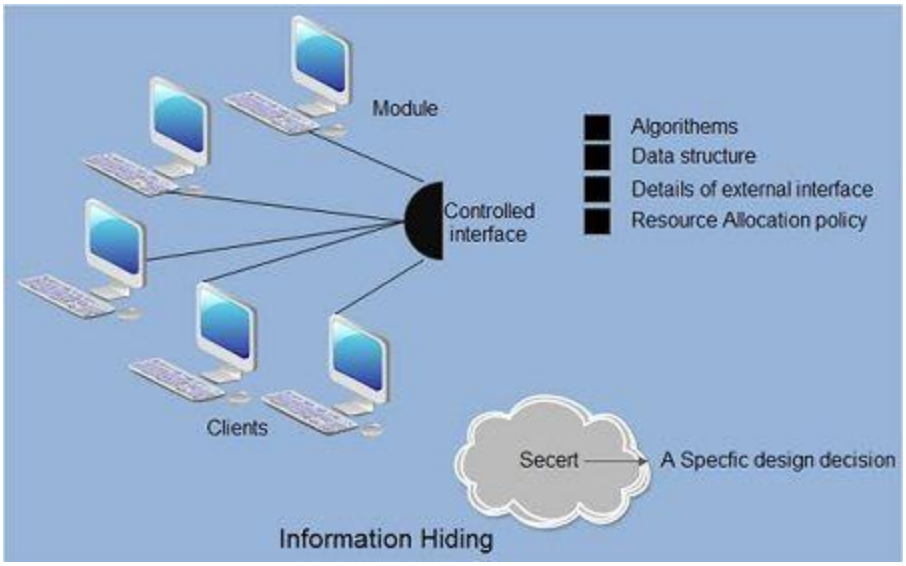
Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.




Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

## Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules. They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as **information hiding. IEEE** defines information hiding as 'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a 'black box' to the other modules in the system.




Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below.

1. Leads to low coupling
2. Emphasizes communication through controlled interfaces
3. Decreases the probability of adverse effects
4. Restricts the effects of changes in one component on others
5. Results in higher quality software.

## Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier

compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises input, process, and output.

1. INPUT
   - Get user's name (string) through a prompt.
   - Get user's grade (integer from 0 to 100) through a prompt and validate.
2. PROCESS
3. OUTPUT

This is the first step in refinement. The input phase can be refined further as given here.

1. INPUT
   - Get user's name through a prompt.
   - Get user's grade through a prompt.
   - While (invalid grade)

   Ask again:
2. PROCESS
3. OUTPUT

**Note:** Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

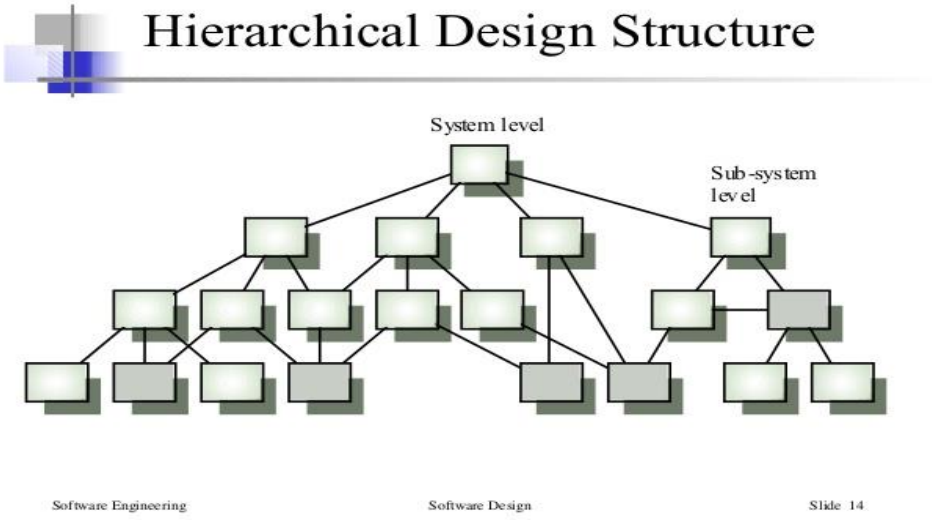
## Refactoring

Refactoring is an important design activity that reduces the complexity of module design keeping its behavior or function unchanged. Refactoring can be defined as a process of modifying a software system to improve the internal structure of design without changing its external behavior. During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc., in order to improve the design. For example, a design model might yield a component which exhibits low cohesion (like a component performs four functions that have a limited relationship with one another). Software designers may decide to refactor the component into four different components, each exhibiting high

cohesion. This leads to easier integration, testing, and maintenance of the software components.

## Control Hierarchy (Program Structure)

- Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control.
- Depth and Width provide an indication of the number of levels of control and overall span of control, respectively.
- Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in specifies how many modules directly control a given a module
- A module that controls another module is said to be superordinate to it, and conversely, a module controlled by another is said to be subordinate to the controller

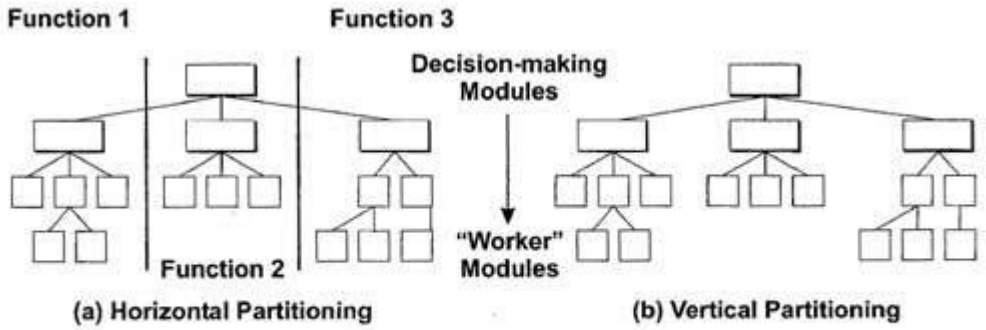



## Structural Partitioning

When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically. In **horizontal partitioning,** the control modules are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits.

- The testing and maintenance of software becomes easier.
- The negative impacts spread slowly.
- The software can be extended easily.

Besides these advantages, horizontal partitioning has some disadvantage also. It requires to pass more data across the module interface, which makes the control flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.




Horizontal and Vertical Partitioning

In **vertical partitioning**, the functionality is distributed among the modules--in a top-down manner. The modules at the top level called **control modules** perform the decision-making and do little processing whereas the modules at the low level called **worker modules** perform all input, computation and output tasks.

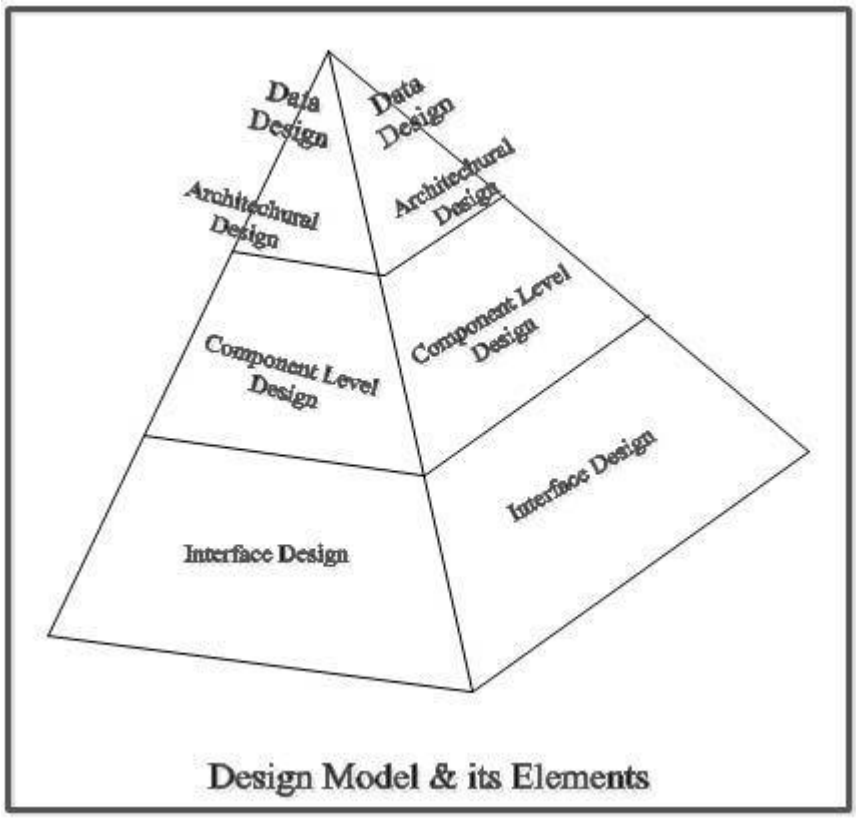
## Data Structure

- Data structure is a representation of the logical relationship among individual elements of data. Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information.
- The organization and complexity of a data structure are limited only by the skill of the designer.
- Limited number of classic data structures that form the building blocks for more sophisticated structures.
- Scalar Items, vectors, and spaces may be organized in a variety of formats

## Developing a Design Model

To develop a complete specification of design (design model), four design models are needed. These models are listed below.

1. **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.
2. **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.
3. **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.
4. **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.




Design Model & its Elements

## Effective Modular Design:

A modular design reduces complexity, facilitates change, and results in easier implementation by encouraging parallel development of different parts of a system.

## Functional Independence

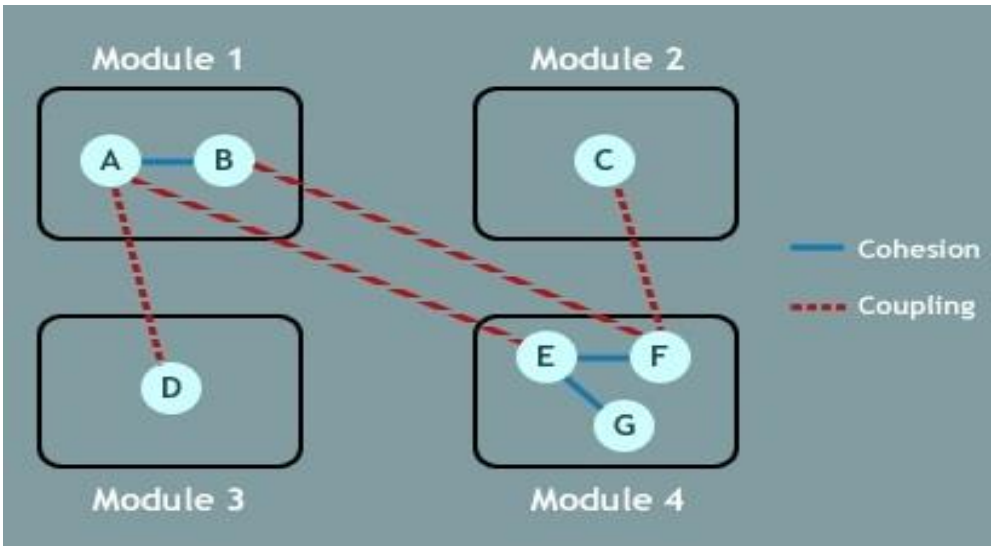
- The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- Independence is measured using two qualitative criteria: cohesion and coupling.
- A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

## Need for functional independence

Functional independence is a key to any good design due to the following reasons:

- **Error isolation**: Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly affect the other modules.
- **Scope of reuse**: Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability**: Complexity of the design is reduced, because different modules can be understood in isolation, as modules are more or less independent of each other.

## Cohesion and coupling:




## Coupling

- Coupling is a measure of interconnection among modules in a software structure.
- In software engineering, the coupling can be defined as the measurement to which the components of the software depend upon each other.
- Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

### Types of Coupling:

- **Data Coupling:**

  If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data.
- **Stamp Coupling:**

  In stamp coupling, the complete data structure is passed from one module to another module.
- **Control Coupling:**

  If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different

behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.

- **External Coupling:**

  In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

- **Common Coupling:** The modules have shared data such as global data structures.

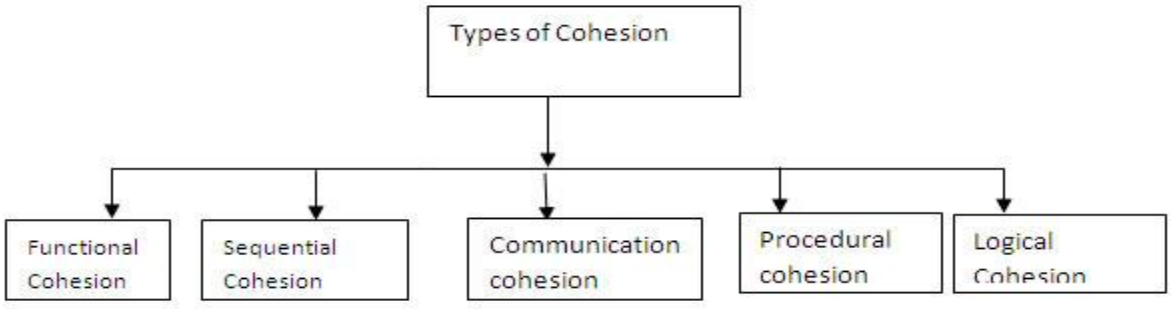
- **Content Coupling:**

  In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

## Cohesion:

- Cohesion is a measure of the relative functional strength of a module.
- Cohesion can be defined as the degree of the closeness of the relationship between its components. In general, it measures the relationship strength between the pieces of functionality within a given module in the software programming.
- It is described as low cohesion or high cohesion.

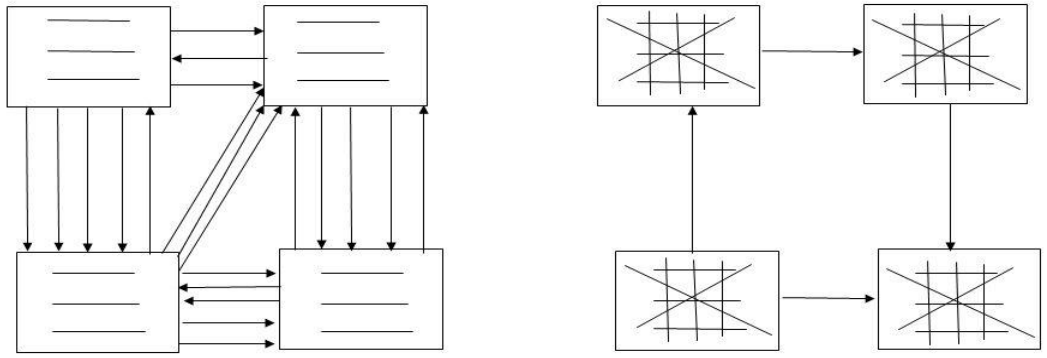
**Types of Cohesion:** There are many different types of cohesion in the software engineering. Some of them are worst, while some of them are best. We have defined them below:



1. **Functional Cohesion:** It is best type of cohesion, in which parts of the module are grouped because they all contribute to the module's single well defined task.
2. **Sequential Cohesion:** When the parts of modules grouped due to the output from the one part is the input to the other, and then it is known as sequential cohesion.
3. **Communication Cohesion:** In Communication Cohesion, parts of the module are grouped because they operate on the same data. For e.g. a module operating on same information records.
4. **Procedural Cohesion:** In Procedural Cohesion, the parts of the module are grouped because a certain sequence of execution is followed by them.
5. **Logical Cohesion:** When the module's parts are grouped because they are categorized logically to do the same work, even though they are all have different nature, it is known as Logical Cohesion. It is one of the worst type of the cohesion in the software engineering.

## Design Principle – Coupling and Cohesion

### Examples of Coupling and Cohesion



High Coupling
Low Cohesion

Low Coupling
High Cohesion

Which one is better from a software design point of view and why?

34

# Difference between coupling and cohesion

| Cohesion | Coupling |
|---|---|
| **Cohesion** is the indication of the relationship within **module**. | **Coupling** is the indication of the relationships between modules. |
| Cohesion shows the module's relative **functional** strength. | Coupling shows the relative **independence** among the modules. |
| Cohesion is a degree (quality) to which a component / module focuses on the **single** thing. | Coupling is a degree to which a component / module is connected to the **other** modules. |
| While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system. | While designing you should strive for **low coupling** i.e. **dependency** between modules should be less. |
| Cohesion is the kind of natural extension of data hiding for example, **class** having all members visible with a package having default visibility. | Making private fields, private methods and non public classes provides loose coupling. |
| Cohesion is **Intra – Module** Concept. | Coupling is **Inter -Module** Concept. |

## Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

OR

Software testing is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.
- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

## Objectives of testing

- Executing a program with the intent of finding an *error*.

- To check if the system meets the requirements and be executed successfully in the Intended environment.
- To check if the system is "Fit for purpose".
- To check if the system does what it is expected to do.
- A good test case is one that has a probability of finding an as yet undiscovered error.
- A successful test is one that uncovers a yet undiscovered error.
- A good test is not redundant.
- A good test should be "best of breed".
- A good test should neither be too simple nor too complex

## Testing Principles:

If you were to test the entire possible combinations project EXECUTION TIME & COSTS would rise exponentially. We need certain principles and strategies to optimize the testing effort. Here are the 7 Principles:

### 1) Exhaustive testing is not possible

- Testing all the functionalities using all valid and invalid inputs and preconditions is known as Exhaustive testing.
- Why it's impossible to achieve Exhaustive Testing?
- Assume we have to test an input field which accepts age between 18 to 20 so we do test the field using 18,19,20. In case the same input field accepts the range between 18 to 100 then we have to test using inputs such as 18, 19, 20, 21, …., 99, 100. It's a basic example, you may think that you could achieve it using automation tool. Imagine the same field accepts some billion values. It's impossible to test all possible values due to release time constraints.
- If we keep on testing all possibletest conditions then the software execution time and costs will rise. So instead of doing exhaustive testing, risks and priorities will be taken into consideration whilst doing testing and estimating testing efforts.

### 2) Defect Clustering

- Defect clustering which states that a small number of modules contain most of the defects detected. As per the Pareto Principle (80-20 Rule), 80% of issues comes from 20% of modules and remaining 20% of issues from remaining 80% of modules. So we do emphasize testing on the 20% of modules where we face 80% of bugs.
- By experience, you can identify such risky modules. But this approach has its own problems
- If the same tests are repeated over and over again, eventually the same test cases will no longer find new bugs.

### 3) Pesticide Paradox

- Repetitive use of the same pesticide mix to eradicate insects during farming will over time lead to the insects developing resistance to the pesticide Thereby ineffective of pesticides on insects.
- Pesticide Paradox in software testing is the process of repeating the same test cases again and again, eventually, the same test cases will no longer find new bugs.
- To overcome this, the test cases need to be regularly reviewed & revised, adding new & different test cases to help find more defects.

### 4) Testing shows a presence of defects

- Testing talks about the presence of defects and don't talk about the absence of defects. I.e. Software Testing reduces the probability of undiscovered defects remaining in the software but even if no defects are found, it is not a proof of correctness.
- Testing shows the presence of defects in the software. The goal of testing is to make the software fail. Sufficient testing reduces the presence of defects. In case testers are unable to find defects after repeated regression testing doesn't mean that the software is bug-free.

- But what if, you work extra hard, taking all precautions & make your software product 99% bug-free. And the software does not meet the needs & requirements of the clients.
- This leads us to our next principle, which states that- Absence of Error

### 5) Absence of Error - fallacy

- It is possible that software which is 99% bug-free is still unusable. This can be the case if the system is tested thoroughly for the wrong requirement.
- Software testing is not mere finding defects, but also to check that software addresses the business needs. The absence of Error is a Fallacy i.e. Finding and fixing defects does not help if the system build is unusable and does not fulfill the user's needs & requirements.
- To solve this problem, the next principle of testing states that Early Testing

### 6) Early Testing

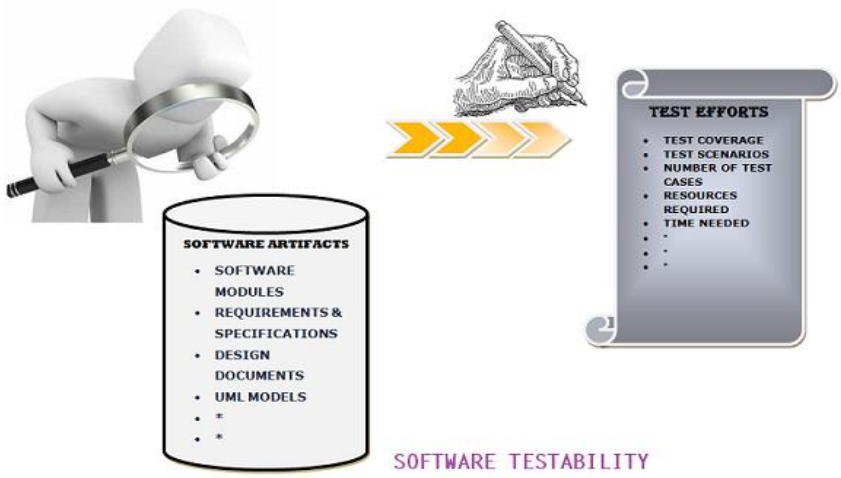
- Defects detected in early phases of SDLC are less expensive to fix. So conducting early testing reduces the cost of fixing defects and defects in the requirements or design phase are captured in early stages.
- It is much cheaper to fix a Defect in the early stages of testing. But how early one should start testing?
- It is recommended that you start finding the bug the moment the requirements are defined. More on this principle in a later training tutorial.
- Assume two scenarios, first one is you have identified an incorrect requirement in the requirement gathering phase and the second one is you have identified a bug in the fully developed functionality. It is cheaper to change the incorrect requirement compared to fixing the fully developed functionality which is not working as intended

### 7) Testing is context dependent

- Testing is context dependent which basically means that the way you test an e-commerce site will be different from the way you test a commercial off the shelf application.
- All the developed software's are not identical. You might use a different approach, methodologies, techniques, and types of testing depending upon the application type. For instance, testing, any POS system at a retail store will be different than testing an ATM machine.

## What is Software Testability?

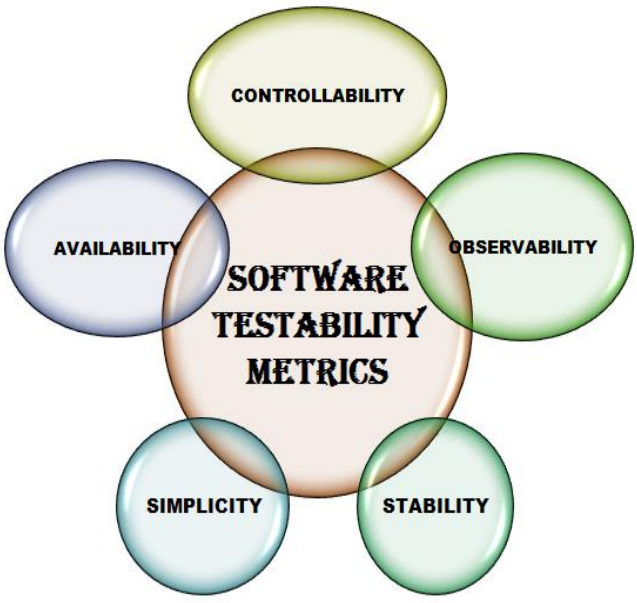
- It is the state of software artifact, which decides the difficulty level for carrying out testing activities on that artifact.
- These software artifacts may include software modules, UML models, requirements & design documents and software application itself.
- It helps in determining the efforts that will be needed to execute test activities on a particular software artifact based on its testability.
- Efforts may be considered in terms of inputs required in testing such as coverage of testing scenarios, the number of test cases, time duration, test resources, etc.
- It is, basically the fundamental attribute associated with each and every software artifact that enables to compute amount of effort required in performing testing. Lesser the testability, larger will be the efforts whereas greater testability ensures minimal efforts.




## How to measure Software's Testability?

Below given are some of the heuristics through which software testability can be determined

- **Controllability:** It defines the control over software and hardware behaviour and comp. Testers should be able to control each module or layers of the software, independently. The better our control, the more effective will be testing.



- **Observability:** You can't test unless & until you perceive something. You can only test, what is visible to you. It is related to observation of states and factors affecting the output of the software.
- **Availability:** It defines the availability of the objects or entities, to carry out the testing. This may include software product evolution at various stages of development, bugs in the software, access to the source code, etc.
- **Simplicity:** The simplicity makes everything easier to use. Lesser efforts are required in testing the simple software product. The simplicity of software may depend upon its functional, structural and code simplicity.
- **Stability:** Lesser the changes better will be the testing. The software product should be stable enough and does not require frequent modification in it. It also examines that changes (if any needed), should be controlled and communicated.

## Key Benefits of Software Testability

- Provides ease to test engineers, to estimate the difficulty, in exploring the defects in the software product.
- Decides the scope of **automated testing** on the software product, based on its controllability.
- Increased in testability ensures the easy and early detection of bugs, thereby saves both time and cost.
- Calculates, minimizes and control the efforts needed by the testers, to perform testing.

## Test Case

- A **TEST CASE** is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.
- The process of developing test cases can also help find problems in the requirements or design of an application.

- Test Case Template- A test case can have the following elements. Note, however, that a test management tool is normally used by companies and the format is determined by the tool used.

| | |
|---|---|
| **Test Suite ID** | The ID of the test suite to which this test case belongs. |
| **Test Case ID** | The ID of the test case. |
| **Test Case Summary** | The summary / objective of the test case. |
| **Related Requirement** | The ID of the requirement this test case relates/traces to. |
| **Prerequisites** | Any prerequisites or preconditions that must be fulfilled prior to executing the test. |
| **Test Procedure** | Step-by-step procedure to execute the test. |
| **Test Data** | The test data, or links to the test data, that are to be used while conducting the test. |
| **Expected Result** | The expected result of the test. |
| **Actual Result** | The actual result of the test; to be filled after executing the test. |
| **Status** | Pass or Fail. Other statuses can be 'Not Executed' if testing is not performed and 'Blocked' if testing is blocked. |
| **Remarks** | Any comments on the test case or test execution. |
| **Created By** | The name of the author of the test case. |
| **Date of Creation** | The date of creation of the test case. |
| **Executed By** | The name of the person who executed the test. |

| | |
|---|---|
| **Date of Execution** | The date of execution of the test. |
| **Test Environment** | The environment (Hardware/Software/Network) in which the test was executed. |

## Test Case Example / Test Case Sample

| | |
|---|---|
| **Test Suite ID** | TS001 |
| **Test Case ID** | TC001 |
| **Test Case Summary** | To verify that clicking the Generate Coin button generates coins. |
| **Related Requirement** | RS001 |
| **Prerequisites** | 1. User is authorized.<br>2. Coin balance is available. |
| **Test Procedure** | 1. Select the coin denomination in the Denomination field.<br>2. Enter the number of coins in the Quantity field.<br>3. Click Generate Coin. |
| **Test Data** | 1. Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5<br>2. Quantities: 0, 1, 5, 10, 20 |
| **Expected Result** | 1. Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5)<br>2. A message 'Please enter a valid quantity between 1 and 10' should be displayed if the specified quantity is invalid. |

| | |
|---|---|
| **Actual Result** | 1. If the specified quantity is valid, the result is as expected.<br>2. If the specified quantity is invalid, nothing happens; the expected message is not displayed |
| **Status** | Fail |
| **Remarks** | This is a sample test case. |
| **Created By** | John Doe |
| **Date of Creation** | 01/14/2020 |
| **Executed By** | Jane Roe |
| **Date of Execution** | 02/16/2020 |
| **Test Environment** | • OS: Windows Y<br>• Browser: Chrome N |

Writing Good Test Cases

- As far as possible, write test cases in such a way that you test only one thing at a time. Do not overlap or complicate test cases. Attempt to make your test cases 'atomic'.
- Ensure that all positive scenarios AND negative scenarios are covered.
- Language:
    - Write in simple and easy-to-understand language.
    - Use active voice instead of passive voice: Do this, do that.
    - Use exact and consistent names (of forms, fields, etc).
- Characteristics of a good test case:
    - *Accurate*: Exacts the purpose.
    - *Economical*: No unnecessary steps or words.

- *Traceable*: Capable of being traced to requirements.
- *Repeatable*: Can be used to perform the test over and over.
- *Reusable*: Can be reused if necessary.

## Test case design techniques

The main purpose of test case design techniques is to test the functionalities and features of the software with the help of effective test cases. The test case design techniques are broadly classified into three major categories.

1. Specification-Based techniques
2. Structure-Based techniques
3. Experience-Based techniques

### 1. Specification-Based or Black-Box techniques

This technique leverages the external description of the software such as technical specifications, design, and client's requirements to design test cases. The technique enables testers to develop test cases that provide full test coverage. The Specification-based or black box test case design techniques are divided further into 5 categories. These categories are as follows:

- Boundary Value Analysis (BVA)
- Equivalence Partitioning (EP)
- Decision Table Testing
- State Transition Diagrams
- Use Case Testing

### 2. Structure-Based or White-Box techniques

The structure-based or white-box technique design test cases based on the internal structure of the software. This technique exhaustively tests the developed code. Developers who have complete information of the software

code, its internal structure, and design help to design the test cases. This technique is further divided into five categories.

- Statement Testing & Coverage
- Decision Testing Coverage
- Condition Testing
- Multiple Condition Testing
- All Path Testing

**3.Experience-Based techniques**

These techniques are highly dependent on tester's experience to understand the most important areas of the software. The outcomes of these techniques are based on the skills, knowledge, and expertise of the people involved. The types of experience-based techniques are as follows:

Error Guessing

In this technique, the testers anticipate errors based on their experience, availability of data and their knowledge of product failure. Error guessing is dependent on the skills, intuition, and experience of the testers.

**Exploratory Testing**

This technique is used to test the application without any formal documentation. There is minimum time available for testing and maximum for test execution. In this, the test design and test execution are performed concurrently.

The successful application of test case design techniques will render test cases that ensure the success of software testing.

# Risk Management

## Definition of Risk

- Risk is an expectation of loss, a potential problem that may or may not occur in the future.
- It is generally caused due to lack of information, control or time.
- A possibility of suffering from loss in software development process is called a software risk.
- Loss can be anything, increase in production cost, development of poor quality software, not being able to complete the project on time.
- A risk is a potential problem – it might happen and it might not
- Conceptual definition of risk
  - – Risk concerns future happenings
  - – Risk involves change in mind, opinion, actions, places, etc. – Risk involves choice and the uncertainty that choice entails

**Two characteristics of risk**

– Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)

– Loss – the risk becomes a reality and unwanted consequences or losses occur

**A software risk can be of two types**

(1) internal risks that are within the control of the project manager and

(2) external risks that are beyond the control of project manager

## Risk Categorization – Approach #1

- **Project risks**
  - – They threaten the project plan
  - – If they become real, it is likely that the project schedule will slip and that costs will increase
- **Technical risks**
  - – They threaten the quality and timeliness of the software to be produced

  - If they become real, implementation may become difficult or impossible
- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product
- Sub-categories of Business risks
  - **Market risk** – building an excellent product or system that no one really wants
  - **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
  - **Sales risk** – building a product that the sales force doesn't understand how to sell
  - **Management risk** – losing the support of senior management due to a change in focus or a change in people
  - **Budget risk** – losing budgetary or personnel commitment

## Risk Categorization – Approach #2

- **Known risks**
  - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- **Predictable risks**
  - Those risks that are extrapolated from past project experience (e.g., past turnover)
- **Unpredictable risks**
  - Those risks that can and do occur, but are extremely difficult to identify in advance

## Risk Management

- Risk management is carried out to:

- Identify the risk
- Reduce the impact of risk
- Reduce the probability or likelihood of risk
- Risk monitoring

## Steps for Risk Management

1) Identify possible risks; recognize what can go wrong
2) Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3) Rank the risks by probability and impact
   - Impact may be negligible, marginal, critical, and catastrophic
4) Develop a contingency plan to manage those risks having high probability and high impact

## Risk Identification

A method for recognizing risks is to create item checklist. In the risk identification step, the team systematically enumerates as many project risks as possible to make them explicit before they become problems. The checklist is used for risk identification and focus is at the subset of known and predictable risk in the following categories:

**1-Product size**

- risks associated with overall size of the software to be built

**2-Business impact**

- risks associated with constraints imposed by management or the marketplace

**3-Customer characteristics**

- risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner

**4-Process definition**

- risks associated with the degree to which the software process has been defined and is followed

**5-Development environment**

- risks associated with availability and quality of the tools to be used to build the project

**6-Technology to be built**

- risks associated with complexity of the system to be built and the "newness" of the technology in the system

**7-Staff size and experience**

- risks associated with overall technical and project experience of the software engineers who will do the work

## Risk Components and Drivers

- The project manager identifies the risk drivers that affect the following risk components
  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of four impact levels
  - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

## Risk Projection (Estimation)

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real

  - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps (see next slide)
- The intent of these steps is to consider risks in a manner that leads to prioritization
- By prioritizing risks, the software team can allocate limited resources where they will have the most impact

**Risk Projection/Estimation Steps**

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Note the overall accuracy of the risk projection so that there will be no misunderstandings

**Risk table**: A risk table provides a project manager with a simple technique for risk projection. It consists of five columns

1. Risk Summary – short description of the risk
2. Risk Category – one of seven risk categories (slide 12)
3. Probability – estimation of risk occurrence based on group input
4. Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
5. RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

| Risk Summary | Risk Category | Probability | Impact (1-4) | RMMM |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |

**Developing a Risk Table**

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk
- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

**Assessing Risk Impact**

- Three factors affect the consequences that are likely if a risk does occur
  - Its nature – This indicates the problems that are likely if the risk occurs
  - Its scope – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - Its timing – This considers when and for how long the impact will be felt
- The overall risk exposure formula is RE = P x C
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

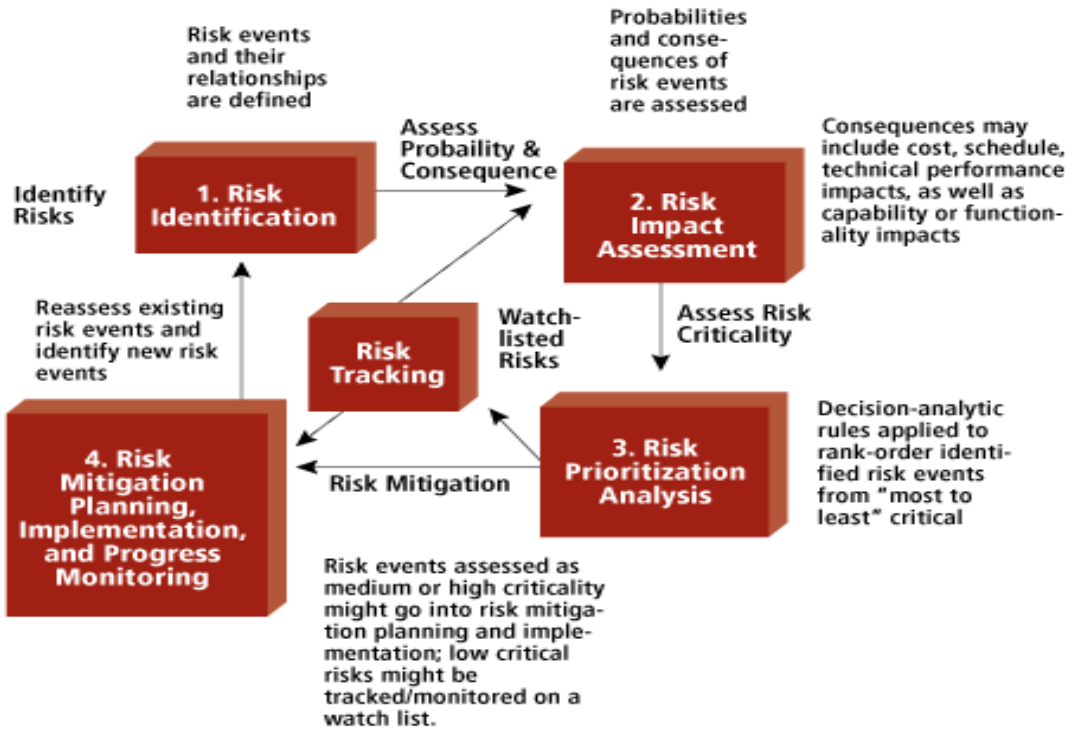
## Risk Mitigation, Monitoring, and Management(RMMM)

R The RMMM plan may be a part of the software development plan or may be a separate document. Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin

- Risk mitigation is a problem avoidance activity
- Risk monitoring is a project tracking activity

Risk analysis support the project team in constructing a strategy to deal with risks. There are three important issues considered in developing an effective strategy:

- **Risk avoidance or mitigation** - It is the primary strategy which is fulfilled through a plan. It is also called as risk control.
- **Risk monitoring** - The project manager monitors the factors and gives an indication whether the risk is becoming more or less.
- **Risk management and planning** - It assumes that the mitigation effort failed and the risk is a reality.




*Risk Management: Fundamental Steps*

Risk mitigation (avoidance) is the primary strategy and is achieved through a plan.

Example: Risk of high staff turnover

Risk mitigation handling options include:

- Assume/Accept: Acknowledge the existence of a particular risk, and make a deliberate decision to accept it without engaging in special efforts to control it. Approval of project or program leaders is required.
- Avoid: Adjust program requirements or constraints to eliminate or reduce the risk. This adjustment could be accommodated by a change in funding, schedule, or technical requirements.
- Control: Implement actions to minimize the impact or likelihood of the risk.
- Transfer: Reassign organizational accountability, responsibility, and authority to another stakeholder willing to accept the risk.
- Watch/Monitor: Monitor the environment for changes that affect the nature and/or the impact of the risk.

**Risk Monitoring**

- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk monitoring has three objectives
    - To assess whether predicted risks do, in fact, occur
    - To ensure that risk aversion steps defined for the risk are being properly applied
    - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

**Risk management** and contingency planning assume that mitigation efforts have failed and that the risk has become a reality

- RMMM steps incur additional project cost
    - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
    - Risks can occur after the software has been delivered to the user

## Seven Principles of Risk Management:

| | |
|---|---|
| **Maintain a global perspective** | View software risks within the context of a system and the business problem that is intended to solve |
| **Take a forward-looking view** | Think about risks that may arise in the future; establish contingency plans |
| **Encourage open communication** | Encourage all stakeholders and users to point out risks at any time |
| **Integrate risk management** | Integrate the consideration of risk into the software process |
| **Emphasize a continuous process of risk management** | Modify identified risks as more becomes known and add new risks as better insight is achieved |
| **Develop a shared product vision** | A shared vision by all stakeholders facilitates better risk identification and assessment |
| **Encourage teamwork when managing risk** | Pool the skills and experience of all stakeholders when conducting risk management activities |