Arba Minch University

Arba Minch Institute of Technology

Faculty of Computing and Software Engineering

Fundamental of Software Engineering (SENG1061)

Learners Material

By: Mikiyas B.

Arba Minch, Ethiopia

March, 2023

Table of Contents

Chapter 1: Introduction 1

1.1 Software 1

1.2 Software Engineering 1

1.3 Good Attribute of Software Engineering 3

1.4 Software Engineering Diversity 3

1.5 Software Engineering Ethics 5

1.6 Challenges of Software Engineering 5

1.7 Solutions to Overcome Challenges in Software Engineering 7

Chapter 2: Software Process 9

2.1 Software Process 9

2.2 Software Process Model 10

2.3 Needs of Software Process Model 11

2.4 Software Process Models 11

2.5 Software Engineering Institute Capability Maturity Model (SEICMM) 61

Chapter 3: Requirement Engineering 65

3.1 Software Requirement 65

3.2 Characteristics of Good Requirement 66

3.3 Requirement Engineering Process 66

3.4 Requirements management 76

3.5 System Model 77

3.5.1 Unified Modelling Language 77

3.6 Characteristics of a Good Requirement 85

Chapter 4: Project Management 86

4.1 Project 86

4.2 Software Project Management 86

4.3 Project Management Framework/ Knowledge Areas 87

4.4 Activities in Project Management 89
4.4.1 Project Planning 89
4.4.2 Project Scheduling 92
Chapter 5: Software Design 94
5.1 Software Design 94
5.1.1 Software Design Levels 95
5.2 Objective of Software Design 96
5.3 Stages of Design 96
5.4 Design Process 96
5.4.1 Software Design Activities 97
5.4.2 Software Design Activities 98
5.5 Functional independence (Coupling and Cohesion) 103
5.6 Software Design Strategies/ Methodology 108
5.6.1 Object Oriented Design System design activities: From objects to subsystems ..110
5.6.2 Software Design Approaches 117
5.7 Principles of Software Design 118
5.8 Architecture Style 122
5.8.1 Pipe-Filter 122
5.8.2 Client-Server 124
5.8.3 Peer-to-Peer 125
5.8.4 Event-Based Architecture 127
5.8.5 Layered Style 128
5.8.6 Model-View-Controller 129
Chapter 6: Software Testing 131
6.1 Software Testing Terminology 131
6.2 Observation about Testing 133
6.3 Objective of Testing 134

6.4 Fundamental principles of testing 135
6.5 Levels of Software of Testing 135
6.5.1 Unit Testing 136
6.5.2 Integration Testing 141
6.5.3 System Testing 142
6.5.4 Acceptance Testing 149
6.6 Classification of software Testing Techniques 151
6.6.1 White Box Strategies 151
6.6.2 Black Box Strategies 152
6.6.3 Grey Box Strategies 152
Bibliography 154

Chapter 1: Introduction

Chapter 1 is a general introduction that introduces professional software engineering and defines some software engineering concepts.

The objectives of this chapter are to introduce software engineering

- Understand what software engineering is and why it is important
- Understand that the development of different types of software systems may require different software engineering techniques
- Understand some ethical and professional issues that are important for software engineers;

1.1 Software

Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market. **Software** may be developed for a particular customer or may be developed for **products** or a general market

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

1.2 Software Engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases.

1. Engineering discipline Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.
2. All aspects of software production Software engineering are not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and efficiently on real machines.

- Software engineering is a **modeling activity.**
- Software engineering is a **problem-solving activity.**
- Software engineering is a **knowledge acquisition activity.**
- Software engineering is a **rationale-driven activity**.

Modeling Activity:

A model is an abstract representation of a system that enables us to answer questions about the system. Software engineers deal with complexity through modeling, by focusing at any one time on only the relevant details and ignoring everything else. Models are useful when dealing with systems that are too large, too small, too complicated, or too expensive

Problem solving

Engineering is a problem-solving activity. It is not algorithmic. In its simplest form, the engineering method includes five steps:

1. formulate the problem
2. analyze the problem
3. search for solutions
4. decide on the appropriate solution
5. specify the solution

Knowledge acquisition

- In modeling the application and solution domain, software engineers collect data, organize it into information, and formalize it into knowledge.
- Knowledge acquisition is **nonlinear,** as a single piece of data can invalidate complete models.
- A common mistake that software engineers and managers make is to assume that the acquisition of knowledge needed to develop a system is linear

Rationale management

- When acquiring knowledge and making decisions about the system or its application domain, software engineers also need to capture the context in which decisions were made and the rationale behind these decisions.
- Enables software engineers to understand the implication of a proposed change when revisiting a decision.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use.

1.3 Good Attribute of Software Engineering

Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.

- **Maintainability**
  - Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
- **Dependability**
  - Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
- **Efficiency**
  - Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
- **Usability**
  - Software must be usable by the users for which it was designed

1.4 Software Engineering Diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. How this systematic approach is actually? implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years. Perhaps the most significant factor in determining which software engineering methods and techniques are

most important is the type of application that is being developed. There are many different types of application including:

1. **Stand-alone applications**: These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.
2. **Interactive transaction-based applications**: These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.
3. **Embedded control systems:** These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
4. **Batch processing systems:** These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.
5. **Entertainment systems**: These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.
6. **Systems for modelling and simulation**: These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.
7. **Data collection systems*:*** These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing. The software has

to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.

8. **Systems of systems** these are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

1.5 Software Engineering Ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer. It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behaviour are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. ***Confidentiality***: You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. **Competence**: You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. **Intellectual property rights**: You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
4. **Computer misuse**: You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

1.6 Challenges of Software Engineering

Every software engineer faces various challenges when it comes to being an excellent programmer. Some of these challenges are particular to the person and their developmental stage. However, regardless of their level of training and expertise, all engineers face similar problems. Let's check the most common software engineer challenges and discuss their responsibilities. These software engineering challenges must be addressed immediately to minimize their effect on your final product.

1. **Legacy systems:** Old, valuable systems must be maintained and updated. For the IT sector, every technological innovation is a blessing. The rapid advancement of technology puts additional pressure on software development professionals to take advantage of these trends when creating new software products to stand out from the crowd and obtain a competitive advantage. It is one of the major software engineering problems.
2. **Heterogeneity**: Systems are distributed and include a mix of hardware and software
3. **Delivery**: There is increasing pressure for faster delivery of software
4. **Increasing customer demands in the development stage:** The majority of software projects are conceptual in nature and are focused on creating software solutions that satisfy a range of consumer needs. Even the simplest application or product requires developers to fully grasp the underlying business concept and incorporate the necessary functionality to meet the ever-increasing client needs. It is among the software engineer coding challenges faced in software development.
5. **Undefined system boundaries:** There might not be a clear set of prerequisites for implementation. The customer might add additional unrelated and irrelevant functionalities on top of the crucial ones, which would result in a very high implementation cost that might go beyond the predetermined budget.
6. **Proper documentation, proper meetings time, and budget constraints:** Confirm your grasp of the requirement by creating a clear requirement document. The aims, scope, limitations, and functional requirements of a product or software program are made clear to teams and individual stakeholders through documentation.

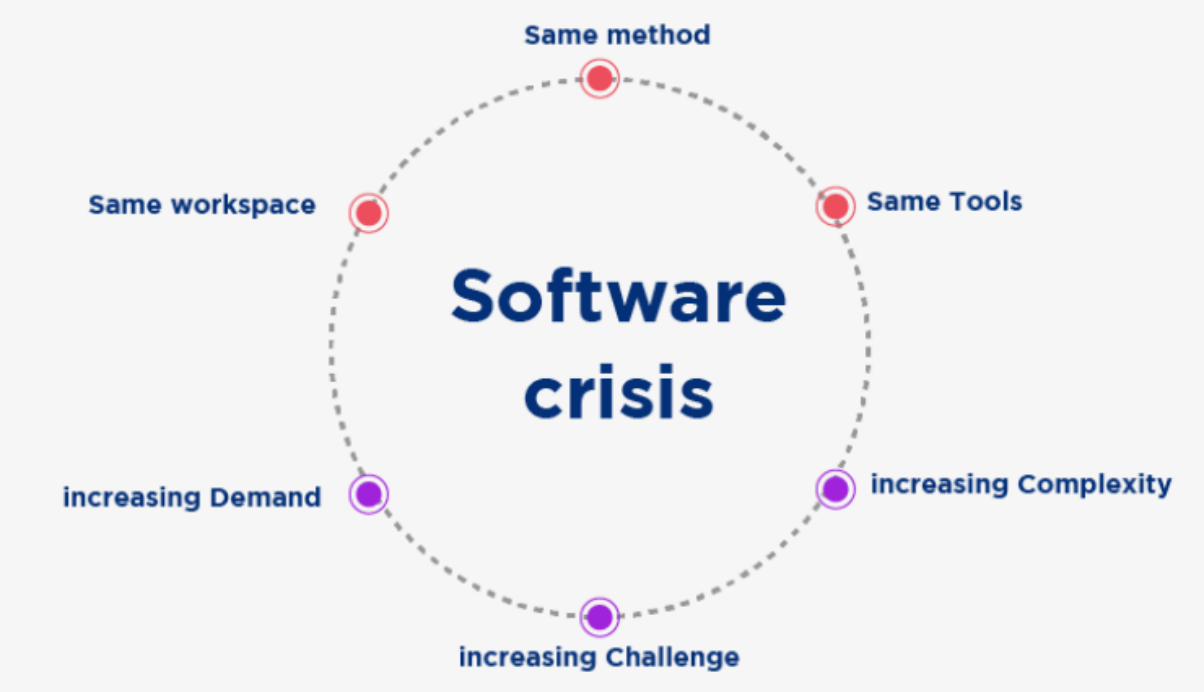


Figure 1: Software Crisis

1.7 Solutions to Overcome Challenges in Software Engineering

We may simplify our life as a software engineer with the aid of these basic fixes. Let's check some of the software engineering solutions for the problems discussed.

1. **Attempting to understand from the viewpoint of stakeholders**

When stakeholders are concerned about the problems and understand that their ideas, opinions, and contributions are respected, software engineering projects are successful. There is evidence to suggest that the way in which the project manager incorporates stakeholders in the decision-making process across the various stages of the project is closely related to the effective execution of software engineering projects.

2. **Recognizing conflicting requirements from the stakeholder side**

It is important to correctly examine and prioritize requirements. Keeping a healthy balance between the requirements and only accepting those that are valid and for which good solutions can be offered within the constraints of the project timeline and budget will help to avoid conflicts.

3. **Creating informative and well-structured conversations with end consumers**

Poor conversations are a major issue for novice engineers. Imagine becoming stuck when coding and being unable to communicate the problem to your team, which could have an impact

on project cost, time, and productivity. Before a project starts, specify the methods and frequency of conversation

4. Performing proper market research and competitors' analysis

A target audience has been identified, comprising the people who are or will use your product or service and their demands. This is one of the key outcomes of the market analysis. A single app cannot serve the needs of all users. If you attempt, you probably won't have any audience at all. Every product is designed for a certain audience, and yours should be as well.

Review Questions

1. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?
2. What are the four important attributes that all professional software should have? Suggest four other attributes that may sometimes be significant.
3. Apart from the challenges of heterogeneity, business and social change, and trust and security, identify other problems and challenges that software engineering is likely to face in the 21st century.
4. Based on your own knowledge of some of the application types discussed, explain with examples, why different application types require specialized software engineering techniques to support their design and development.

Chapter 2: Software Process

The objective of this chapter is to introduce you to the idea of a software Process—a coherent set of activities for software production.

At the end of this chapter student will:

- *understand the concepts of software processes and software process models;*
- *have been introduced to different generic software process models and when they might be used;*
- *know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;*
- *understand why processes should be organized to cope with changes in the software requirements and design;*
- *Understand how process model integrates good software engineering practice to create adaptable software processes.*

2.1 Software Process

The software process defines the way in which software development is organized, managed, measured, supported and improved. (Independently of the type of support technology exploited in the development). It is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components. All organizations involved in software development follow a process of some kind, implicit or explicit, reproducible, instrumented or adaptable.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. **Software specification:** The functionality of the software and constraints on its operation must be defined.
2. **Software design and implementation:** The software to meet the specification must be produced.
3. **Software validation**: The software must be validated to ensure that it does what the customer wants.
4. **Software evolution**: The software must evolve to meet changing customer needs.

In some form, these activities are part of all software processes. In practice, of course, they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There is also supporting process activities such as documentation and software configuration management.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

1. **Products**, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. **Roles**, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
3. **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

2.2 Software Process Model

Software process model is a simplified representation of a software process. A process model (also called software life cycle model) is a descriptive and diagrammatic representation of the software life cycle. Thus, **no matter which process model is followed**, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models

Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. That is, we see the framework of the process but not the details of specific activities. These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

2.3 Needs of Software Process Model

The software development team must have a clear understanding about **when and what to do.**

- Without using of a particular process model
  - The development of a software product would not be in a systematic and disciplined manner.
  - it becomes difficult for software project managers to monitor the progress of the project.
- A **software process model defines entry and exit criteria for every phase**.
  - A phase can start only if its phase-entry criteria have been satisfied.

2.4 Software Process Models

Many software process models have been proposed so far. Each of them has some advantages as well as some disadvantages. A process model for software engineering is chosen based on the

- Nature of project and application
- Methods and tools to be used
- Controls and deliverables that are required

A few important and commonly used Software Process models are as follows:

1. Waterfall Process Model

It suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Also called the classic life cycle or the linear sequential model. It is known as waterfall because it has separate and distinct phases of specification and development. Once a phase is completed you can't repeat it. This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on as shown in Figure 2.

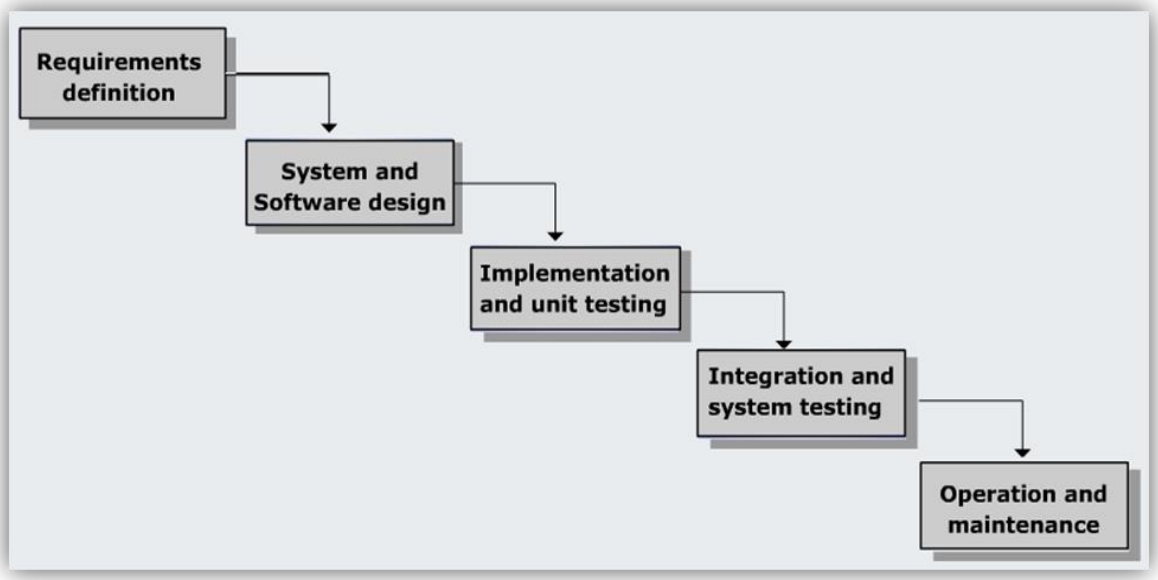


Figure 2: Waterfall Process Model

Phases in Waterfall Model

The principal stages of the waterfall model directly reflect the fundamental development

Activities:

A. **Requirements analysis and definition** the system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
B. **System and software design** the systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
C. **Implementation and unit testing** during this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
D. **Integration and system testing** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
E. **Operation and maintenance** Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving

the implementation of system units and enhancing the system's services as new requirements are discovered.

Pro's and Con's of Waterfall Model

Advantages:

- ✓ Simple to implement and manage.
- ✓ Enforces a disciplined software development approach

Drawbacks:

- ✓ Unable to accommodate changes at later stages
- ✓ A working version of the software is not available during development
- ✓ Unable to accommodate iteration
- ✓ Deadlock can occur due to delay of any step
- ✓ Model is not suitable for large projects

When to Use Waterfall Model

- ✓ Requirements are very well known
- ✓ Product definition is stable
- ✓ Technology is understood
- ✓ New version of an existing product
- ✓ Porting an existing product to a new platform.

2. V-Shaped SDLC Model

A variant of the Waterfall that emphasizes the verification and validation of the product. Testing of the product is planned in parallel with a corresponding phase of development.

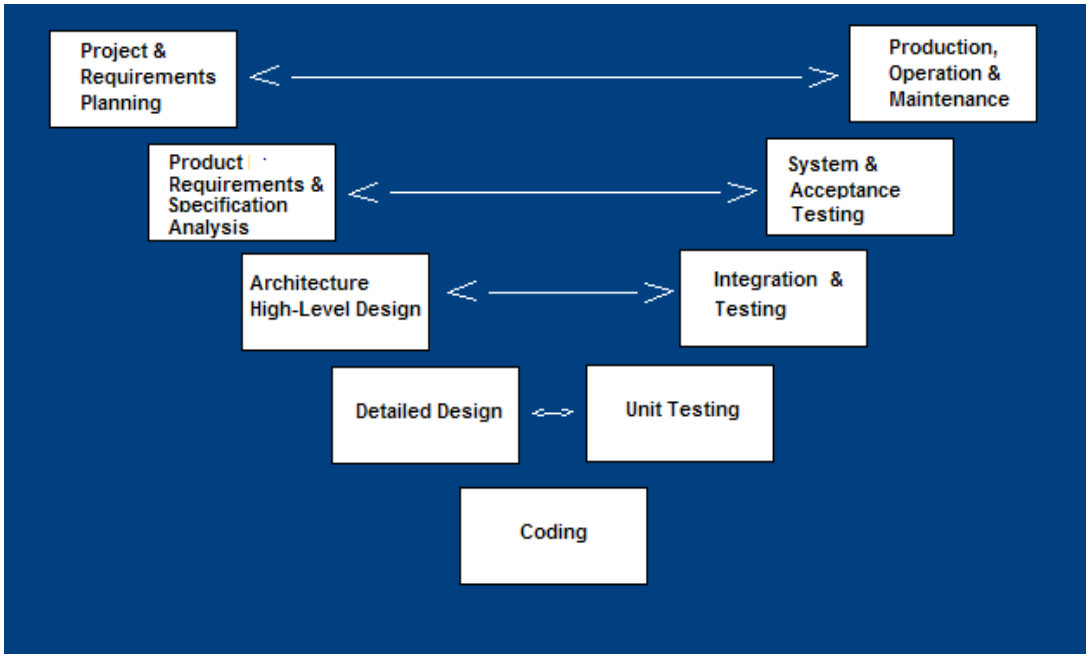


Figure 3: V-Shaped Model

Project and Requirements Planning allocate resources

Product Requirements and Specification Analysis – complete specification of the software system

Architecture or High-Level Design – defines how software functions fulfill the design

Detailed Design – develop algorithms for each architectural component

Production, operation and maintenance provide for enhancement and corrections

System and acceptance testing – check the entire software system in its environment

Integration and Testing – check that modules interconnect correctly

Unit testing – check that each module acts as expected

Coding – transform algorithms into software

V-Shaped SDLC Model Strengths

- ✓ Emphasize planning for verification and validation of the product in early stages of product development
- ✓ Each deliverable must be testable
- ✓ Project management can track progress by milestones
- ✓ Easy to use

V-Shaped SDLC Model Strengths

- ✓ Does not easily handle concurrent events
- ✓ Does not handle iterations or phases
- ✓ Does not easily handle dynamic changes in requirements
- ✓ Does not contain risk analysis activities

When to Use V-Shaped SDLC Model

- ✓ Excellent choice for systems requiring high reliability – hospital patient control applications
- ✓ All requirements are known up-front
- ✓ When it can be modified to handle changing requirements beyond analysis phase
- ✓ Solution and technology are known

3. Structured Evolutionary Prototyping Model

- ✓ Developers build a prototype during the requirements phase
- ✓ Prototype is evaluated by end users
- ✓ Users give corrective feedback
- ✓ Developers further refine the prototype

- ✓ When the user is satisfied, the prototype code is brought up to the standards needed for a final product.

❖ **Structured Evolutionary Prototyping Model Strengths**

- ✓ Customers can "see" the system requirements as they are being gathered
- ✓ Developers learn from customers
- ✓ A more accurate end product
- ✓ Unexpected requirements accommodated
- ✓ Allows for flexible design and development
- ✓ Steady, visible signs of progress produced
- ✓ Interaction with the prototype stimulates awareness of additional needed functionality

❖ **Structured Evolutionary Prototyping Model Weakness**

- ✓ Tendency to abandon structured program development for "code-and-fix" development
- ✓ Bad reputation for "quick" methods
- ✓ Overall maintainability may be overlooked
- ✓ The customer may want the prototype delivered.
- ✓ Process may continue forever (scope creep)

❖ **Structured Evolutionary Prototyping Model Weakness**

- ✓ Requirements are unstable or have to be clarified
- ✓ As the requirements clarification stage of a waterfall model
- ✓ Develop user interfaces
- ✓ Short-lived demonstrations
- ✓ New, original development

- ✓ With the analysis and design portions of object- oriented development.

4. **Rapid Application Model (RAD)**

RAD is a linear sequential software development process model that emphasizes a concise development cycle using an element based construction approach. If the requirements are well understood and described, and the project scope is a constraint, the RAD process enables a development team to create a fully functional system within a concise time period.

❖ **When to use RAD Model**

- ✓ When the system should need to create the project that modularizes in a short span time (2-3 months).
- ✓ When the requirements are well-known.
- ✓ When the technical risk is limited.
- ✓ When there's a necessity to make a system, which modularized in 2-3 months of period.
- ✓ It should be used only if the budget allows the use of automatic code generating tools.

Advantage of RAD Model

- ✓ This model is flexible for change.
- ✓ In this model, changes are adoptable.
- ✓ Each phase in RAD brings highest priority functionality to the customer.
- ✓ It reduced development time.
- ✓ It increases the reusability of features.

Disadvantage of RAD Model

- ✓ It required highly skilled designers.
- ✓ All application is not compatible with RAD.
- ✓ For smaller projects, we cannot use the RAD model.
- ✓ On the high technical risk, it's not suitable.
- ✓ Required user involvement.

5. **Incremental Development**

Incremental Model is a process of software development where requirements divided into multiple standalone modules of the software development cycle. In this model, each module goes through the requirements, design, implementation and testing phases. Every subsequent release of the module adds function to the previous release. The process continues until the complete system achieved.

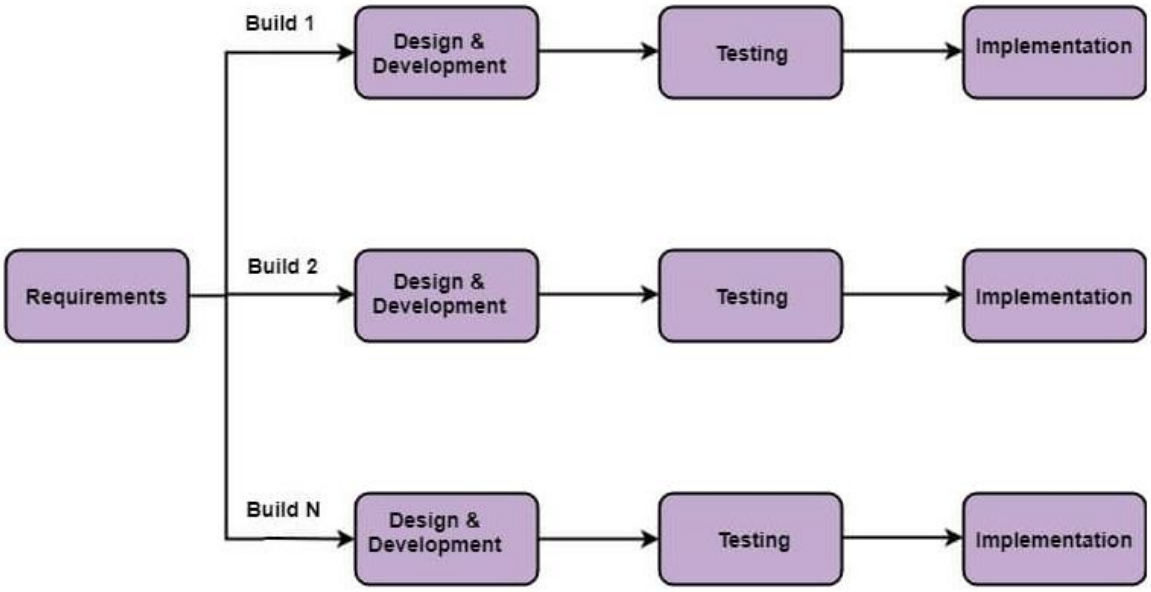


Figure 4: Incremental Model

The various phases of incremental model are as follows:

1. **Requirement analysis:** In the first phase of the incremental model, the product analysis expertise identifies the requirements. And the system functional requirements are understood by the requirement analysis team. To develop the software under the incremental model, this phase performs a crucial role.
2. **Design & Development:** In this phase of the Incremental model of SDLC, the design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.
3. **Testing:** In the incremental model, the testing phase checks the performance of each existing function as well as additional functionality. In the testing phase, the various methods are used to test the behaviour of each task.

4. **Implementation:** Implementation phase enables the coding phase of the development system. It involves the final coding that design in the designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up to the final system product.

❖ **When we use the Incremental Model?**

- ✓ When the requirements are superior.
- ✓ A project has a lengthy development schedule.
- ✓ When Software team are not very well skilled or trained.
- ✓ When the customer demands a quick release of the product.
- ✓ You can develop prioritized requirements first.

❖ **Advantage of Incremental Model**

- ✓ Errors are easy to be recognized.
- ✓ Easier to test and debug
- ✓ More flexible.
- ✓ Simple to manage risk because it handled during its iteration.
- ✓ The Client gets important functionality early.
- ✓ Uses "divide and conquer" breakdown of tasks
- ✓ Customers get important functionality early

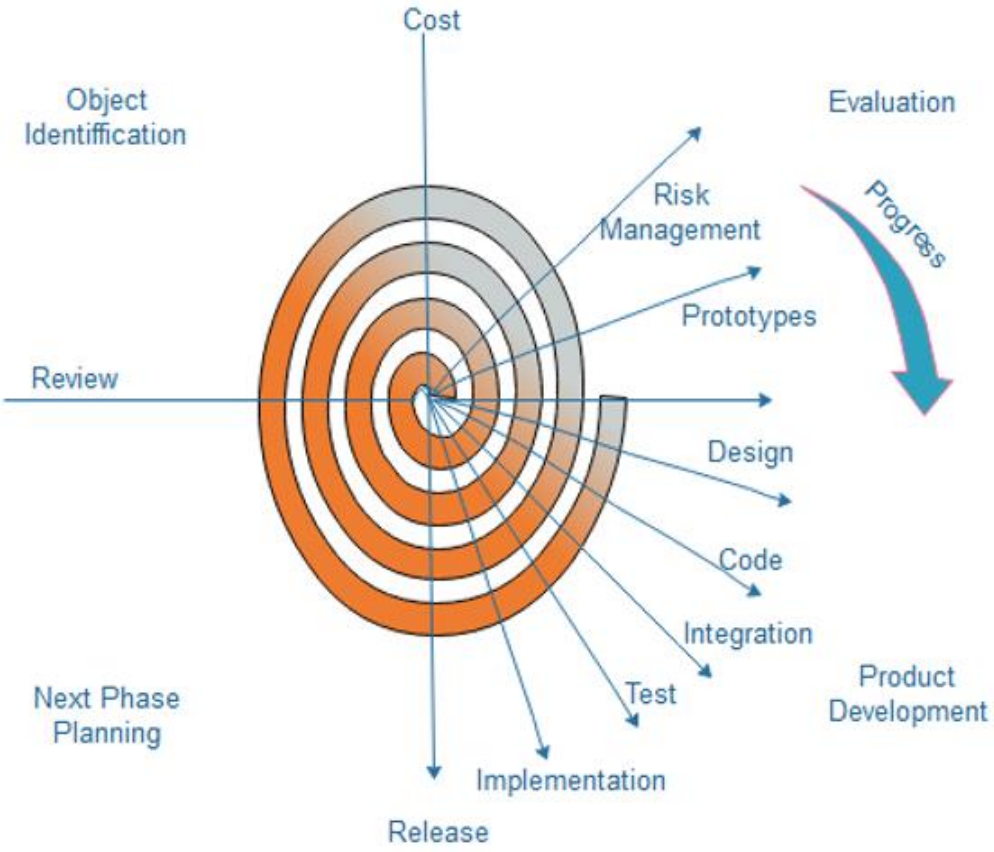
❖ **Disadvantage of Incremental Model**

- ✓ Need for good planning
- ✓ Total Cost is high.
- ✓ Well defined module interfaces are needed.

6. Spiral Process Model

The spiral model, initially proposed by Boehm, is an evolutionary software process model that couples the iterative feature of prototyping with the controlled and systematic aspects of the linear sequential model. It implements the potential for rapid development of new versions of the

software. Using the spiral model, the software is developed in a series of incremental releases. During the early iterations, the additional release may be a paper model or prototype. During later iterations, more and more complete versions of the engineered system are produced.



Each cycle in the spiral is divided into four parts:

Objective setting: Each cycle in the spiral starts with the identification of purpose for that cycle, the various alternatives that are possible for achieving the targets, and the constraints that exists.

Risk Assessment and reduction: The next phase in the cycle is to calculate these various alternatives based on the goals and constraints. The focus of evaluation in this stage is located on the risk perception for the project.

Development and validation: The next phase is to develop strategies that resolve uncertainties and risks. This process may include activities such as benchmarking, simulation, and prototyping.

Planning: Finally, the next step is planned. The project is reviewed, and a choice made whether to continue with a further period of the spiral. If it is determined to keep, plans are drawn up for the next step of the project.

The development phase depends on the remaining risks. For example, if performance or user-interface risks are treated more essential than the program development risks, the next phase may be an evolutionary development that includes developing a more detailed prototype for solving the risks.

The **risk-driven** feature of the spiral model allows it to accommodate any mixture of a specification-oriented, prototype-oriented, simulation-oriented, or another type of approach. An essential element of the model is that each period of the spiral is completed by a review that includes all the products developed during that cycle, including plans for the next cycle. The spiral model works for development as well as enhancement projects.

When to use Spiral Model

- ✓ When deliverance is required to be frequent.
- ✓ When the project is large
- ✓ When requirements are unclear and complex
- ✓ When changes may require at any time
- ✓ Large and high budget projects

Advantages

- ✓ High amount of risk analysis
- ✓ Useful for large and mission-critical projects.
- ✓ Users see the system early because of rapid prototyping tools
- ✓ Critical high-risk functions are developed first
- ✓ The design does not have to be perfect
- ✓ Users can be closely tied to all lifecycle steps
- ✓ Early and frequent feedback from users
- ✓ Cumulative costs assessed frequently

Disadvantages

- ✓ Risk analysis needed highly particular expertise
- ✓ Doesn't work well for smaller projects.
- ✓ Time spent for evaluating risks too large for small or low- risk projects
- ✓ Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- ✓ The model is complex
- ✓ Developers must be reassigned during non-development phase activities
- ✓ May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration

7. The Rational Unified Process (RUP)

RUP is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process. it is a good example of a hybrid process model. It brings together elements from all of the generic process models The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Each cycle corresponds to, for example, a period in which a new generation of a product is being worked on. The Rational Unified Process (RUP) divides development into the four consecutive phases:

- ✓ Inception phase
- ✓ Elaboration phase
- ✓ Construction phase
- ✓ Transition phase

Each phase is finalised with a milestone. A milestone is a point in time where decisions of critical importance must to be made. In order to be able to make those decisions, the objectives must have been accomplished.

Phase 1: Inception

During the first phase, the basic idea and structure of the project are determined. In this phase, the team meets regularly to determine the project's necessity, but also its viability and suitability. Viability and suitability also include the expected costs and the means needed to complete the project after the green light has been given.

Depending on the project, the result of the first phase could be:

- ✓ First use case (20% completed)
- ✓ Market research results
- ✓ Financial prognosis
- ✓ Project plan
- ✓ Corporate or business model
- ✓ Prototypes

Phase 2: Elaboration

During the elaboration phase, the system's requirements and its required architecture are assessed and analysed. This is where the project begins to take shape. The objective of the elaboration phase is to analyse products and to lay a foundation for the future architecture. Results of the elaboration phase include:

- ✓ Use case (80% completed)
- ✓ Description of the feasible architecture
- ✓ Project development plan
- ✓ Prototypes for tackling risks
- ✓ User manual

Criteria for the results:

- ✓ Is the architecture stable?

- ✓ Are important risks being tackled?
- ✓ Is the development plan sufficiently detailed and accurate?
- ✓ Do all interested parties agree on the current design?
- ✓ Are the expenditures acceptable?

Phase 3: Construction

In the construction phase of the Rational Unified Process (RUP), the software system is constructed in its entirety. The emphasis is on the development of components and other features of the system.

The majority of coding also takes place in this phase. In this production process, the emphasis is on managing costs and means, as well as ensuring quality. Results from the production phase include:

- ✓ Fully completed software system
- ✓ User manual

To be assessed according to:

- ✓ Is the product stable and complete enough for use?
- ✓ Are all interested parties/users ready for the transition into the product's usage?
- ✓ Are all the expenditures and means still in good order?

Phase 4: Transition

The objective of the transition phase is to transfer the product to its new user. As soon as the user starts using the system, problems almost always arise that require changes to be made to the system. The goal, however, is to ensure a positive and smooth transition to the user. Results and activities in the last phase:

- ✓ Beta testing
- ✓ Conversion of existing user databases
- ✓ Training new users
- ✓ Rolling out of the project to marketing and distribution

8. Agile Model

The meaning of Agile is swift or versatile. "**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

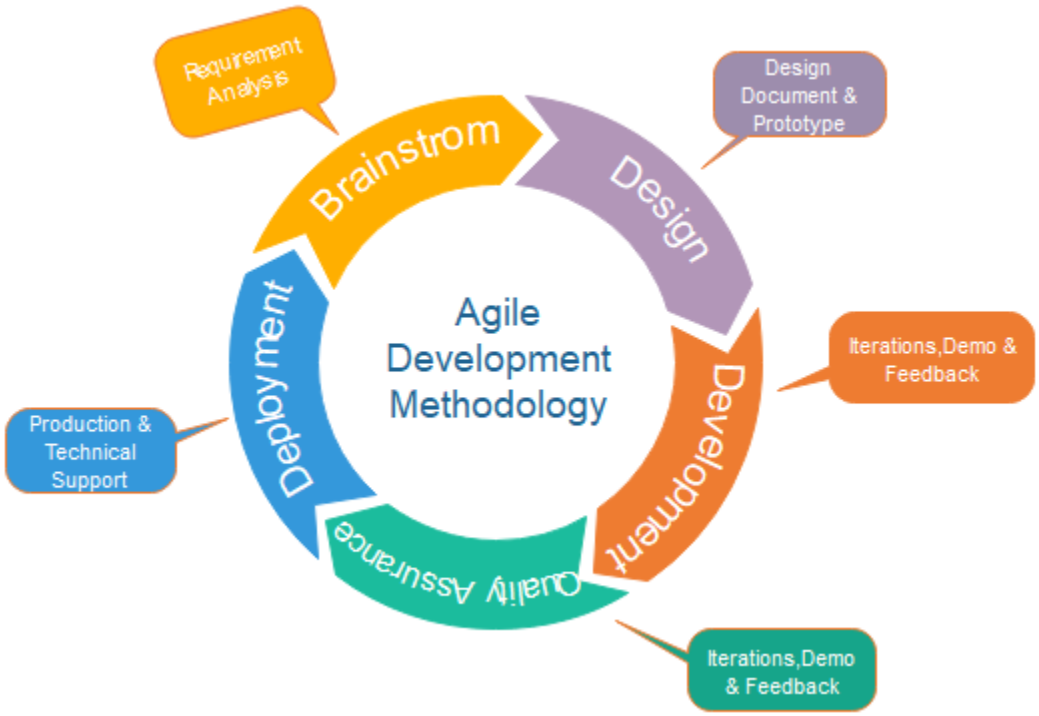


Figure 5: Agile Model

Phases of Agile Model:

Following are the phases in the agile model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration
4. Testing/ Quality assurance
5. Deployment

6. Feedback

1. **Requirements gathering:** In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.
2. **Design the requirements:** When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.
3. **Construction/ iteration:** When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.
4. **Testing:** In this phase, the Quality Assurance team examines the product's performance and looks for the bug.
5. **Deployment:** In this phase, the team issues a product for the user's work environment.
6. **Feedback:** After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback

❖ **When to use the Agile Model?**

1. When frequent changes are required.
2. When a highly qualified and experienced team is available.
3. When a customer is ready to have a meeting with a software team all the time.
4. When project size is small.

❖ **Advantage(Pros) of Agile Method:**

- ✓ Frequent Delivery
- ✓ Face-to-Face Communication with clients.
- ✓ Efficient design and fulfils the business requirement.
- ✓ Anytime changes are acceptable.
- ✓ It reduces total development time.

❖ **Disadvantages(Cons) of Agile Model:**

- ✓ Due to the shortage of formal documents, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
- ✓ Due to the lack of proper documentation, once the project completes and the developers allotted to another project, maintenance of the finished project can become a difficulty.

2.5 Software Engineering Institute Capability Maturity Model (SEICMM)

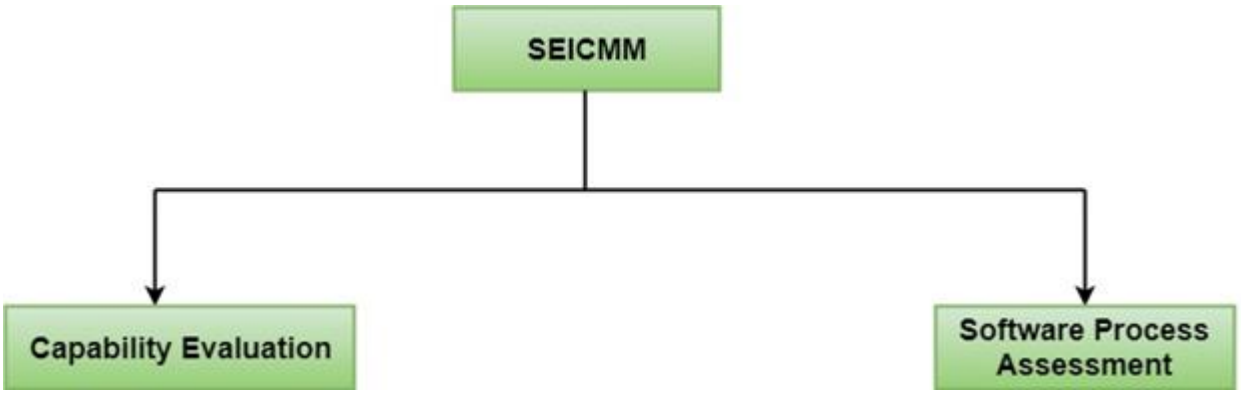
The Capability Maturity Model (CMM) is a procedure used to develop and refine an organization's software development process.

The model defines a five-level evolutionary stage of increasingly organized and consistently more mature processes.

CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development canter promote by the U.S. Department of Defence (DOD).Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

Methods of SEICMM

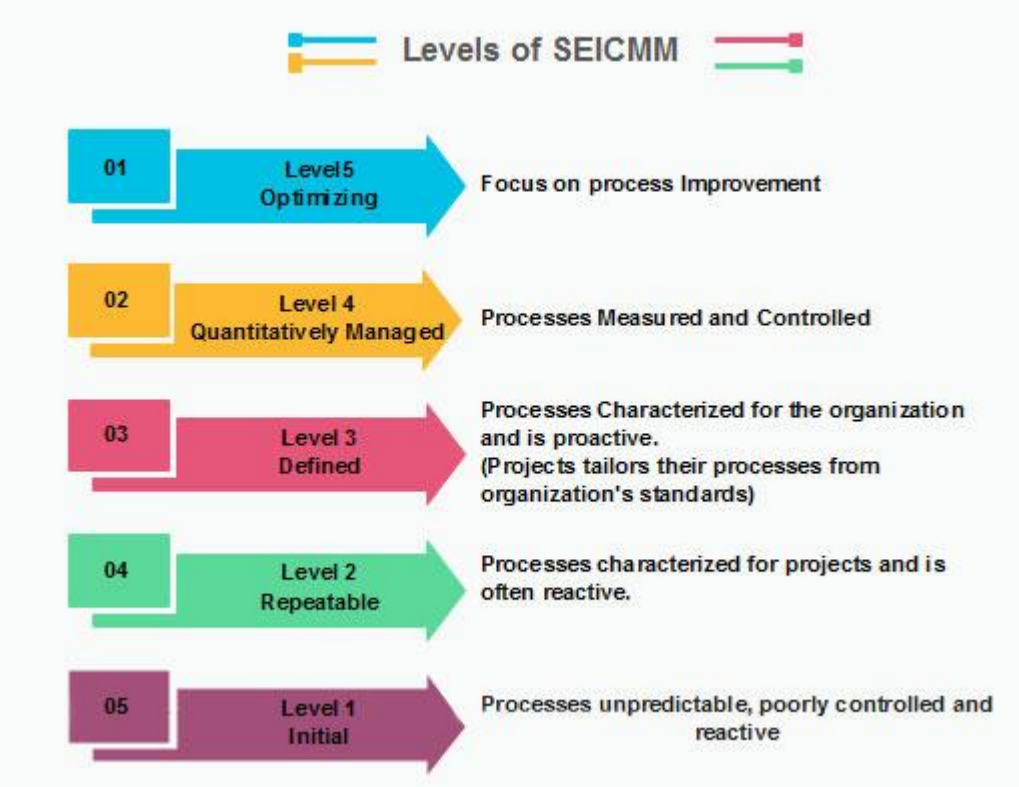
There are two methods of SEICMM:



Capability Evaluation: Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicate the likely contractor performance if the contractor is awarded a work. Therefore, the results of the software process capability assessment can be used to select a contractor.

Software Process Assessment: Software process assessment is used by an organization to improve its process capability. Thus, this type of evaluation is for purely internal use.

SEI CMM categorized software development industries into the following five maturity levels. The various levels of SEI CMM have been designed so that it is easy for an organization to build its quality system starting from scratch slowly.



Level 1: Initial

Ad hoc activities characterize a software development organization at this level. Very few or no processes are described and followed. Since software production processes are not limited, different engineers follow their process and as a result, development efforts become chaotic. Therefore, it is also called a chaotic level.

Level 2: Repeatable

At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.

Level 3: Defined

At this level, the methods for both management and development activities are defined and documented. There is a common organization-wide understanding of operations, roles, and responsibilities. The ways through defined, the process and product qualities are not measured. ISO 9000 goals at achieving this level.

Level 4: Managed

At this level, the focus is on software metrics. Two kinds of metrics are composed. **Product metrics** measure the features of the product being developed, such as its size, reliability, time complexity, understand ability, etc.

Process metrics follow the effectiveness of the process being used, such as average defect correction time, productivity, the average number of defects found per hour inspection, the average number of failures detected during testing per LOC, etc. The software process and product quality are measured, and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to analyze if a project performed satisfactorily. Thus, the outcome of process measurements is used to calculate project performance rather than improve the process.

Level 5: Optimizing

At this phase, process and product metrics are collected. Process and product measurement data are evaluated for continuous process improvement.

Key Process Areas (KPA) of a software organization

Except for SEI CMM level 1, each maturity level is featured by several Key Process Areas (KPAs) that contains the areas an organization should focus on improving its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.

CMM Level	Focus	Key Process Areas
1. Initial	Competent People	NO KPA'S
2. Repeatable	Project Management	Software Project Planning software Configuration Management
3. Defined	Definition of Processes	Process definition Training Program Peer reviews
4. Managed	Product and Process quality	Quantitiative Process Metrics Software Quality Management
5. Optimizing	Continuous Process improvement	Defect Prevention Process change management Technology change management

The focus of each SEI CMM level and the Corresponding Key process areas.

SEI CMM provides a series of key areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a method for gradual quality improvement over various stages. Each step has been carefully designed such that one step enhances the capability already built up.

Chapter 3: Requirement Engineering

The objective of this chapter is to introduce software requirements and to discuss the processes involved in discovering and documenting these requirements. When you have read the chapter you will:

- ✓ *Eliciting requirements from the customers,*
- ✓ *understand the concepts of user and system requirements and why these requirements should be written in different ways*
- ✓ *understand the differences between functional and non-functional software requirements;*
- ✓ *understand how requirements may be organized in a software requirements document;*
- ✓ *understand the principal requirements engineering activities of elicitation, analysis and validation, and the relationships between these activities*
- ✓ *Understand why requirements management is necessary and how it supports other requirements engineering activities.*
- ✓ *Reviewing requirements to ensure their quality,*

3.1 Software Requirement

Requirements are descriptions of the services that a software system must provide and the constraints under which it must operate. The requirements for a system are the descriptions of what the system should do the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analysing, documenting and checking these services and constraints is called requirements engineering (RE). Requirements Engineering is the process of establishing the services that the customer requires from the system and the constraints under which it is to be developed and operated.

The term ‘requirement’ is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function.

Requirements may serve a dual function:

- ✓ As the basis of a bid for a contract
- ✓ As the basis for the contract itself

3.2 Characteristics of Good Requirement

- ✓ Clear and Unambiguous
  - standard structure
  - has only one possible interpretation
  - Not more than one requirement in one sentence
- ✓ Correct
  - A requirement contributes to a real need
- ✓ Understandable
  - A reader can easily understand the meaning of the requirement
- ✓ Verifiable
  - A requirement can be tested
- ✓ Complete
- ✓ Consistent
- ✓ Traceable
- ✓ Making Requirements Testable:
  - Fit criteria form objective standards for judging whether a proposed solution satisfies the requirements:
    - ❖ It is easy to set fit criteria for quantifiable requirements.
    - ❖ It is hard for subjective quality requirements.
  - Three ways to help make requirements testable:
    - ❖ Specify a quantitative description for each adverb and adjective.
    - ❖ Replace pronouns with specific names of entities.
    - ❖ Make sure that every noun is defined in exactly one place in the requirements documents.

3.3 Requirement Engineering Process

The process of establishing what services are required and the constraints on the system's operation and development. Requirements engineering help software engineers to better understand the problem they will work to solve. It encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants and how end-users will interact with the software.

- ✓ **Requirement Engineering Process**
  - Feasibility Study
  - Requirements elicitation and analysis
  - Requirements Specification
  - Requirements Validation

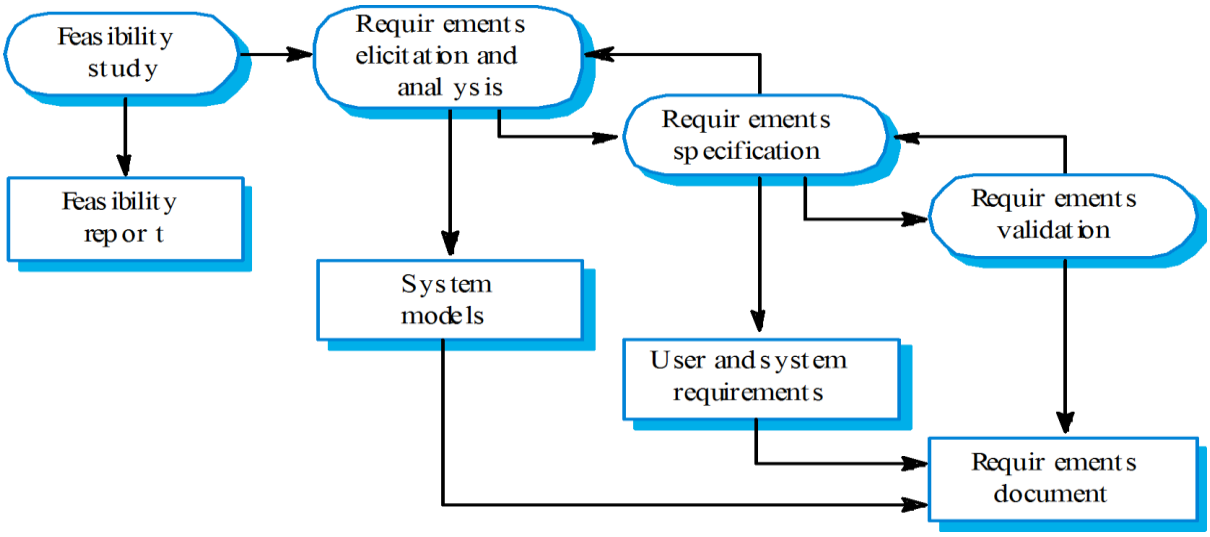


Figure 6: Requirement Engineering Process

1. Feasibility Study

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions:

- ✓ Does the system contribute to the overall objectives of the organization?
- ✓ Can the system be implemented within schedule and budget using current technology?
- ✓ Can the system be integrated with other systems that are used?

Feasibility study an estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2. Requirement Elicitation and Analysis

Requirements elicitation is the practice of obtaining the requirements of a system from users, customers and other stakeholders. The practice is also sometimes referred to as Requirement gathering. It is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.

- ✓ Requirements elicitation practice include the following:
  - **Interviews**
  - **Questionnaires**
  - **User observation**
  - **Workshops**
  - **Brain storming**
  - **Use cases**
  - **Role playing**
  - **And prototyping**

Requirements elicitation includes the followingg activities:

1. Identifying actors. During this activity, developers identify the different types of users the future system will support.
2. Identifying scenarios. During this activity, developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system.

- ✓ scenarios are concrete examples illustrating a single case,

3. Identifying use cases: developers derive from the scenarios a set of use cases that completely represent the future system.

- ✓ use cases are abstractions describing all possible cases.

4. Refining use cases: developers ensure that the system specification is complete, by detailing each use case and describing the behaviour of the system in the presence of errors and exceptional conditions.

5. Identifying relationships among use cases: developers consolidate the use case model by eliminating redundancies. This ensures that the system specification is consistent.
6. Identifying non-functional requirements: developers, users, and clients agree on aspects that are visible to the user but not directly related to functionality. These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

A. Scenarios

People usually find it easier to relate to real-life examples rather than abstract descriptions. They can understand and criticize a scenario of how they might interact with a software system. Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.

Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system. A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction.

Scenarios are real-life examples of how a system can be used.

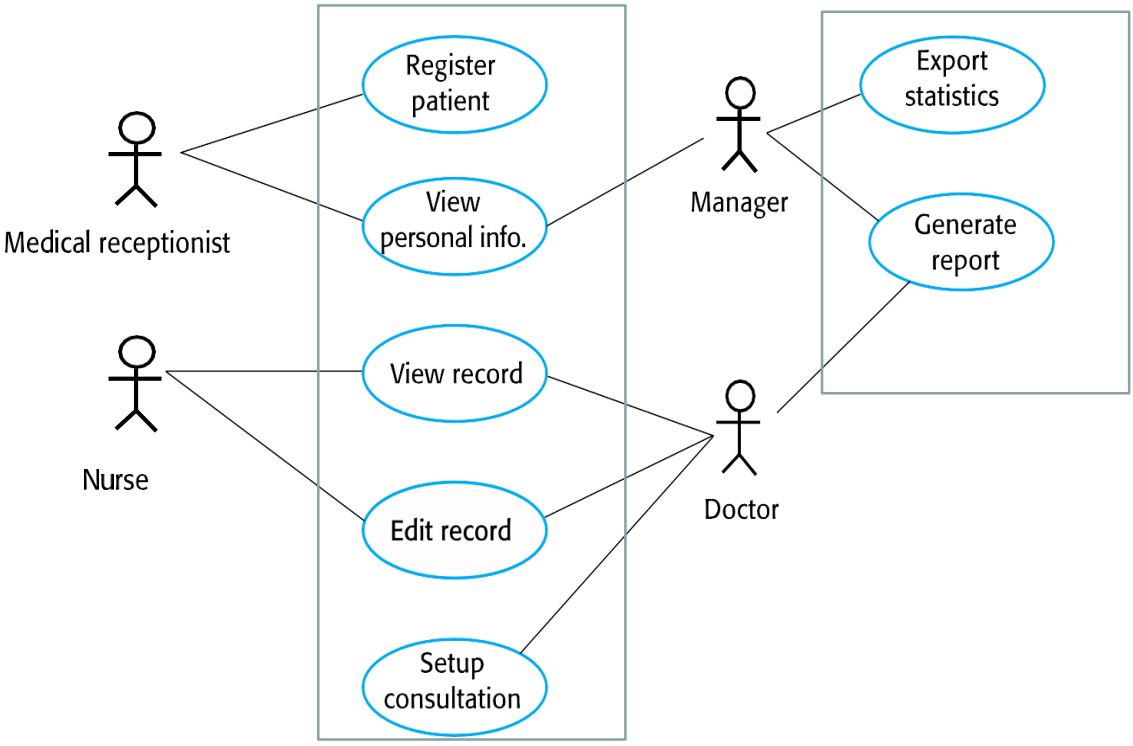
- They should include
  - A description of the starting situation
  - A description of the normal flow of events
  - A description of what can go wrong and how this is handled.
  - Information about other activities that might be going on at the same time.
  - A description of the system state when the scenario finishes.
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

B. Use cases

A scenario-based technique in UML which identify the actors in an interaction and which describe the interaction itself. Used also to clarify the system boundaries. A set of use cases should describe all possible interactions with the system.

Use cases are a requirements discovery technique that were first introduced in the Objectory method [1]. They have now become a fundamental feature of the unified modelling language. In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 3.3, which shows some of the use cases for the patient information system.



❖ **Guidelines of Requirements Elicitation**

- ✓ Assess the business and technical feasibility for the proposed system
- ✓ Identify the people who will help specify requirements.
- ✓ Define the technical environment (e.g. computing architecture, operating system, telecommunication needs) into which the system or product will be placed
- ✓ Identify "domain constraints" (i.e. characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to build
- ✓ Define one or more requirements elicitation methods
- ✓ Solicit participation from many people so that requirements are defined from different point of views.
- ✓ Create usage scenarios of use cases to help customers/ users better identify key requirements

❖ **Problems of Requirement Elicitation**

- ✓ **Problems of scope:** The boundary of system is ill- defined. Or unnecessary details are provided.
- ✓ **Problems of understanding:** The users are not sure of what they need, and don't have full understanding of the problem domain.
- ✓ **Problems of volatility:** the requirements change overtime.
- ✓ Stakeholders don't know what they really want.
- ✓ Stakeholders express requirements in their own terms.
- ✓ Different stakeholders may have conflicting requirements.
- ✓ Organisational and political factors may influence the system requirements.
- ✓ The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

Requirement Analysis

Requirements Analysis, determining whether the stated requirements are clear, complete, consistent and unambiguous.

❖ **Stakeholder Identification**

Stakeholders are people or organizations that have a valid interest in the system. They may be affected by it directly or indirectly. Stakeholders may include:

- ✓ Anyone who operates the system
- ✓ Anyone who benefits from the system
- ✓ Anyone involved in purchasing or procuring the system
- ✓ People opposed to the system (negative)

❖ **Stakeholder Interviews**

- ✓ Interviews are a common technique used in requirement analysis.
- ✓ This technique can serve as a means of obtaining the highly focused knowledge from different stakeholder's perspectives

❖ **Type of Requirement**

✓ **Architectural Requirements:**

A formal description and representation of a system, organized in a way that support reasoning about the structure of the system which comprises system components, the externally visible properties of those components, the relationships and the behaviour between them, and provides a plan from which products can be procured and systems developed, that will work together to implement

✓ **User requirements**

Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

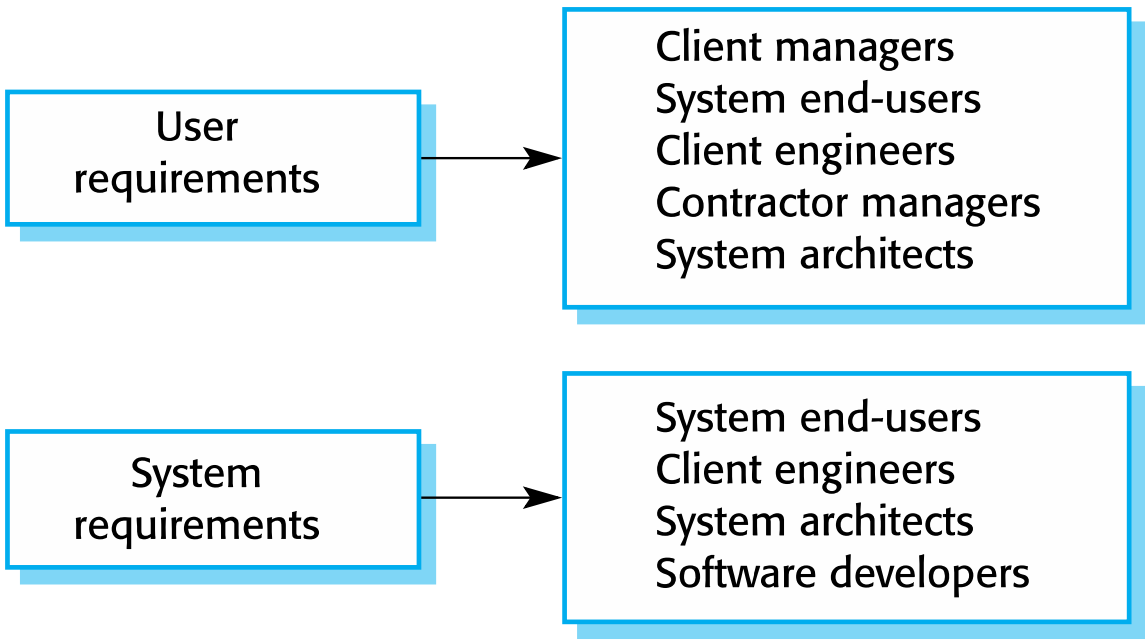
✓ **System requirements**

A structured document setting out detailed descriptions of the system's:

- functions,
- Services,
- operational constraints.

Defines what should be implemented

- So, it may/will be part of a contract between client and contractor.



- ✓ **Functional Requirements:**

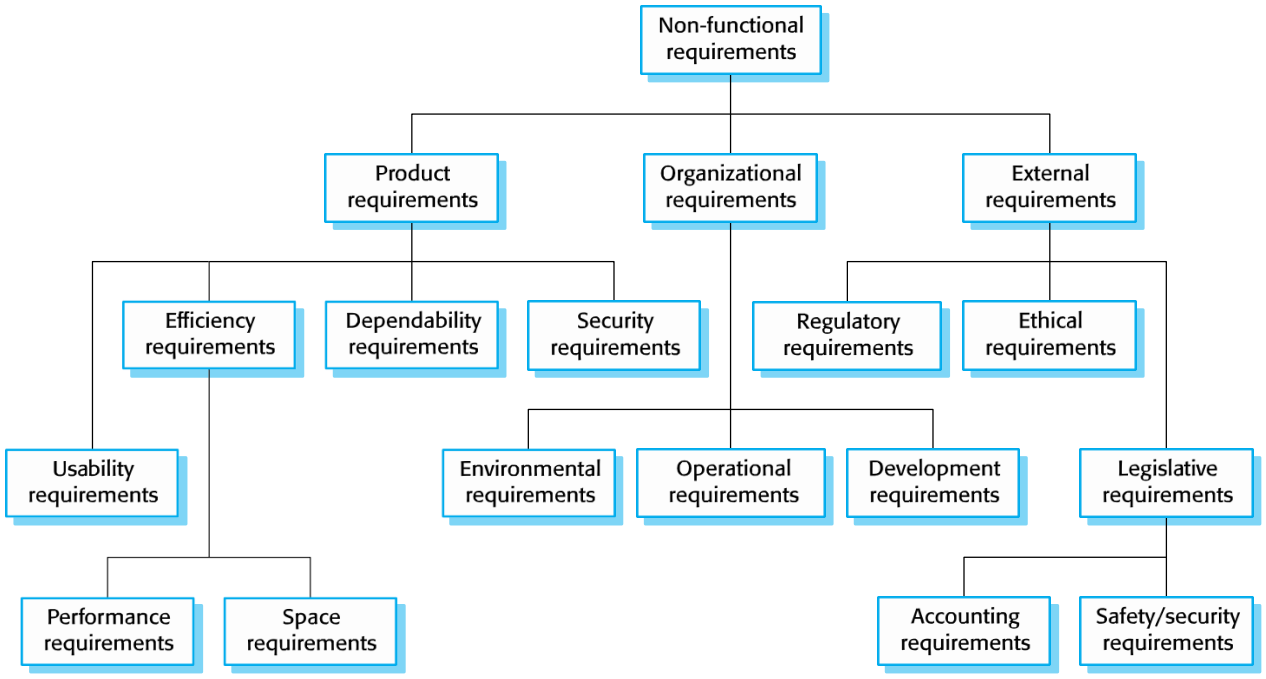
Defines functions of a software system or its components. They may be calculations, technical details, data manipulation and processing and other specific functionality that define "what a system is supposed to accomplish?"

They describe particular results of a system Functional requirement are supported by non-functional requirement

- ✓ **Non-Functional Requirements:**

They are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviour. Functional requirements define what the system is supposed to *do*, whereas non-functional requirements define how a system is supposed to *be*. Non-functional requirements can be divided into two main categories:

- ✓ Execution qualities, such as security and usability, which are observable at runtime.
- ✓ Evolution qualities, such as testability, maintainability and scalability.



Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Metrics for specifying non-functional requirements

3. Requirements Specification

Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. **User requirements** are abstract statements of the system requirements for the customer and end-user of the system; **system requirements** are a more detailed description of the functionality to be provided.

It includes a set of use cases that describe all the interactions the users will have with the software. In addition to use cases, the SRS also contains non- functional requirements (such as performance requirements, quality standards, or design constraints).

❖ A Software Requirements Specification (SRS)

The software requirement specification document enlists all necessary requirements for project development. To derive the requirements, we need to have clear and thorough understanding of the products to be developed.

A general organization of an SRS is as follows:

- ✓ Introduction
  - Purpose, Scope, Definitions, System Overview, References
- ✓ Overall Description
  - Product Perspective, Product functions, User characteristics, constraints, assumptions and dependencies.
- ✓ Specific Requirements
  - External Interface requirements, functional requirements, performance requirements, design constraints, logical database requirement, software system attributes.

4. Requirements Validation and Verification

Validation & Verification), is the process of checking whether the requirements, as identified, do not contradict the expectations about the system of various stakeholders and do not contradict each other. It is Requirements Quality Control.

Validation: "Am I building the right product?" checking a work product against higher-level work products or authorities that frame this particular product.

- ✓ Requirements are validated by stakeholders

Verification: "Am I building the product right?" checking a work product against some standards and conditions imposed on this type of product and the process of its development.

- ✓ Requirements are verified by the analysts mainly

3.4 Requirements management

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing. The system requirements must then also evolve to reflect this changed problem view. Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done. Once end users have experience of a system, they will discover new needs and priorities. There are several reasons why change is inevitable:

Requirements management is the process of managing changing requirements during the requirements engineering process and system development. Requirements are inevitably incomplete and inconsistent

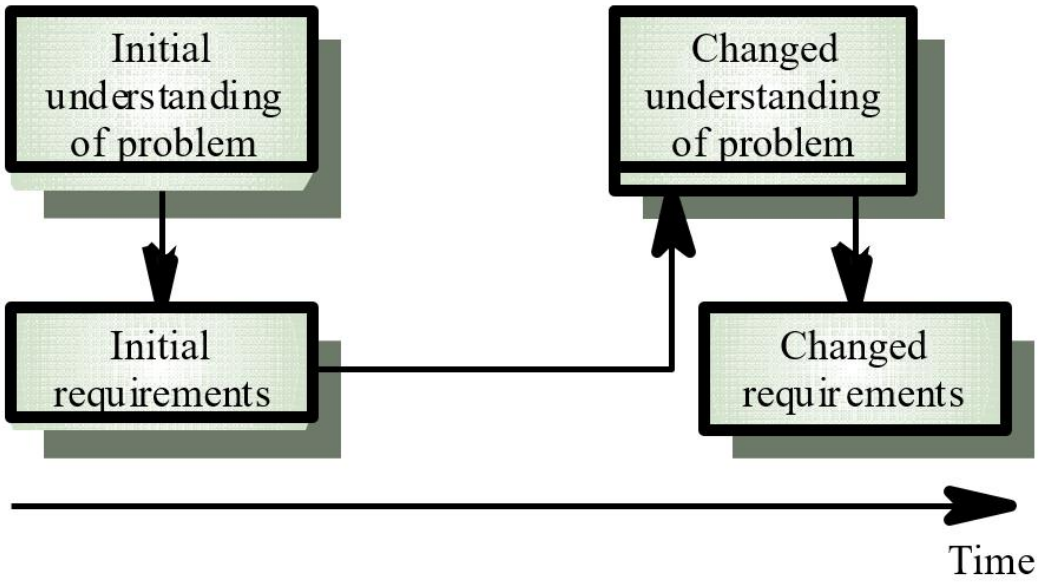
- ✓ New requirements emerge during the process as business needs change and a better understanding of the system is developed
- ✓ Different viewpoints have different requirements and these are often contradictory

Requirement management is a set of activities that help project team to identify, control, and track requirements and changes as project proceeds. Requirements begin with identification. Each requirement is assigned a unique identifier. Once requirement have been identified, traceability table are developed.

Traceability Table:

- ✓ **Features traceability table** - shows how requirements relate to customer observable features

- ✓ **Source traceability table** - identifies source of each requirement
- ✓ **Dependency traceability table** - indicate relations among requirements
- ✓ **Subsystem traceability table** - requirements categorized by subsystem
- ✓ **Interface traceability table** - requirement relations to internal and external interfaces. It will help to track, if change in one requirement will affect different aspects of the system.



3.5 System Model

Different models may be produced during the requirements analysis activity. Requirements analysis may involve three structuring activities which result in these different models

- ✓ **Partitioning:** Identifies the structural (part-of) relationships between entities
- ✓ **Abstraction:** Identifies generalities among entities
- ✓ **Projection:** Identifies different ways of looking at a problem

3.5.1 Unified Modelling Language

It's an industry standard graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system." [2]. UML is a generic broad language enabling the key aspects of software development to capture on paper.

- ✓ General-purpose modeling language
- ✓ It's a notation not a methodology.

System development focuses on three different models of the system:

- ✓ **The functional model:** represented in UML with use case diagrams, describes the **functionality of the system** from the user's point of view.
- ✓ **The object model:** represented in UML with class diagrams, describes the **structure of a system** in terms of objects, attributes, associations, and operations.
- ✓ The **dynamic model:** represented in UML with sequence diagrams, state chart diagrams, and activity diagrams, describes the **internal behavior of the system.**

Benefit of UML

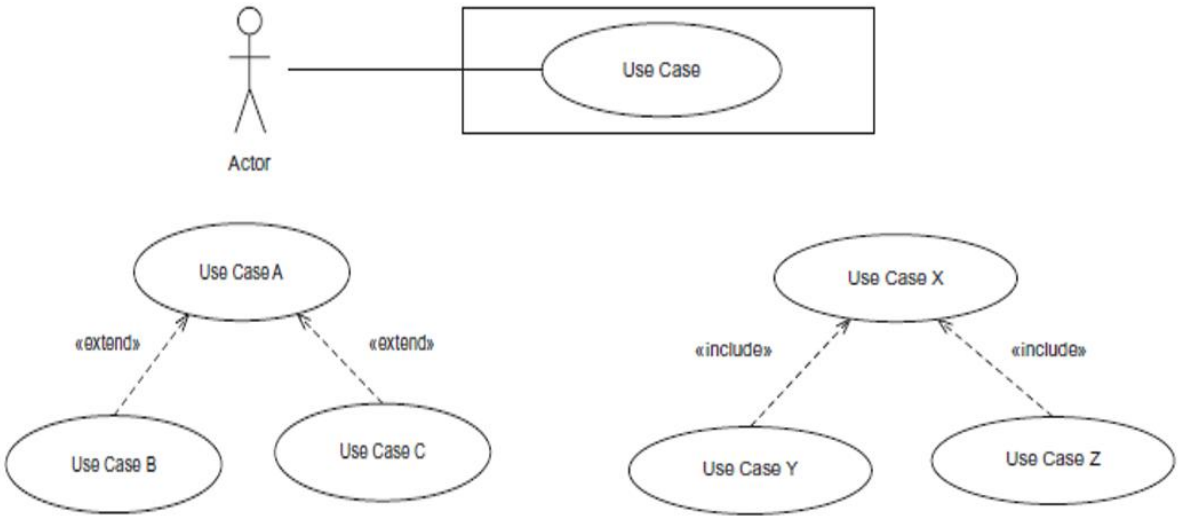
- ✓ Use graphical notation to communicate more clearly than natural language (imprecise) and code (too detailed).
- ✓ Quick understanding
- ✓ Help acquire an overall view of a system.
- ✓ Lessens chances of conflicts.
- ✓ UML is not dependent on any one language or technology.
- ✓ Necessary to manage complexity.
- ✓ UML moves us from fragmentation to standardization.

1. Use case Diagram

A **use case** is the description of set of interaction between an actor and system. Derived from use-case study scenarios. It is an overview of use cases, actors, and their communication relationships to demonstrate how the system reacts to requests from external users. It is used to capture system requirements. Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse with the actors involved in the use case.

Component of Use case

1. **Actor: -** any entity that interacts with the system (e.g., a user, another system, the system's physical environment).
2. **Use case: -** A set of scenarios that describe an interaction between a user and a system, including alternatives
3. **System boundary: -** rectangle diagram representing the boundary between the actors and the system.



❖ **Types of relationships in use case diagram**

1. **Association: -** communication between an actor and a use case.
   - ✓ Represented by a solid line.

2. **Generalization: -** relationship between one general use case and a special use case.
   - ✓ Represented by a line with a triangular arrow head toward the parent use case.

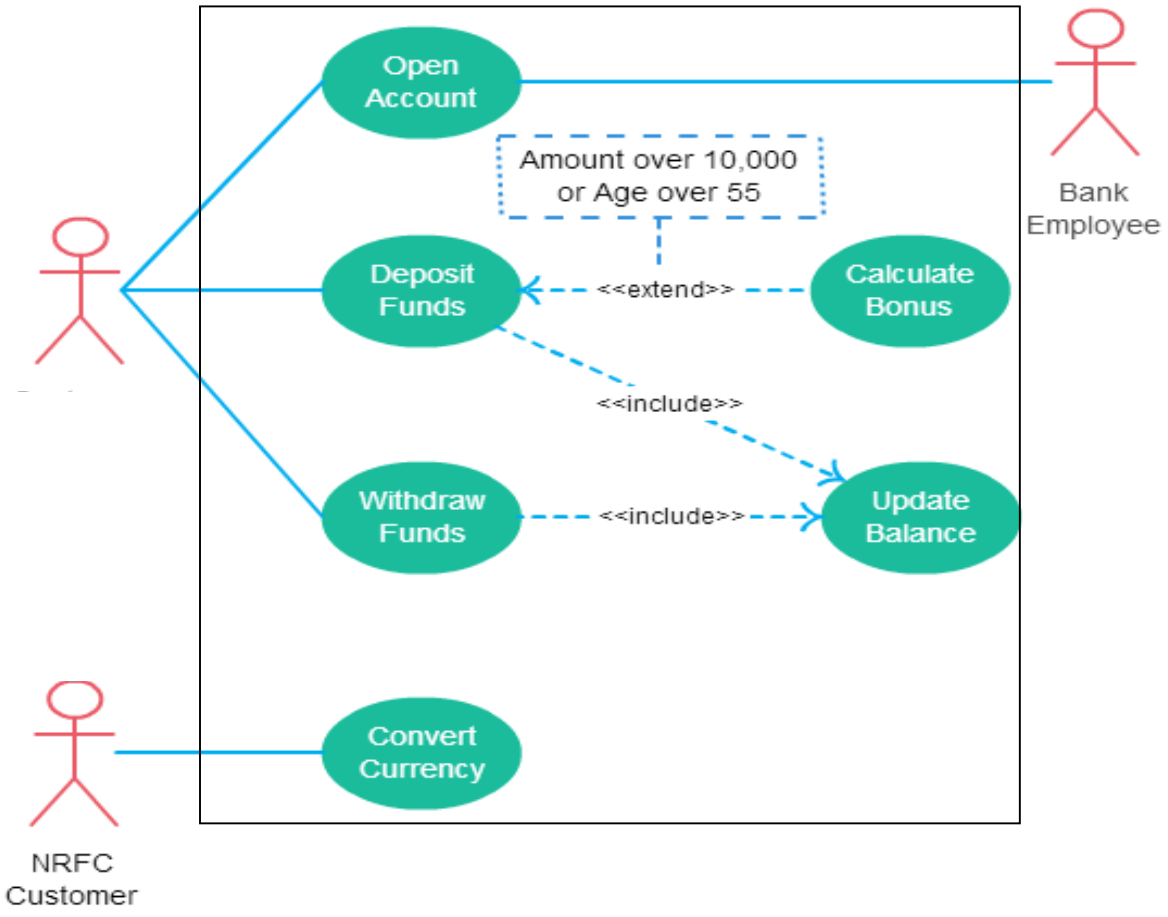
3. **Include: -** occurs when a chunk of behavior is similar across more than one use case.
   - ✓ Use "include" instead of copying the description of that behavior.
   - ✓ The base use case is incomplete without the included use case.
   - ✓ The included use case is mandatory and not optional.

- ✓ The direction of a <<includes>> relationship is to the using use case

<<include>>

4. **Extend: -** The extending use case may add behavior to the base use case.
   - ✓ The direction of an <<extends>> relationship is to the extended use case

<<extend>>



❖ **Use case Description**

Each Use Case contains a full set of textual details about the interactions and scenarios contained within it.

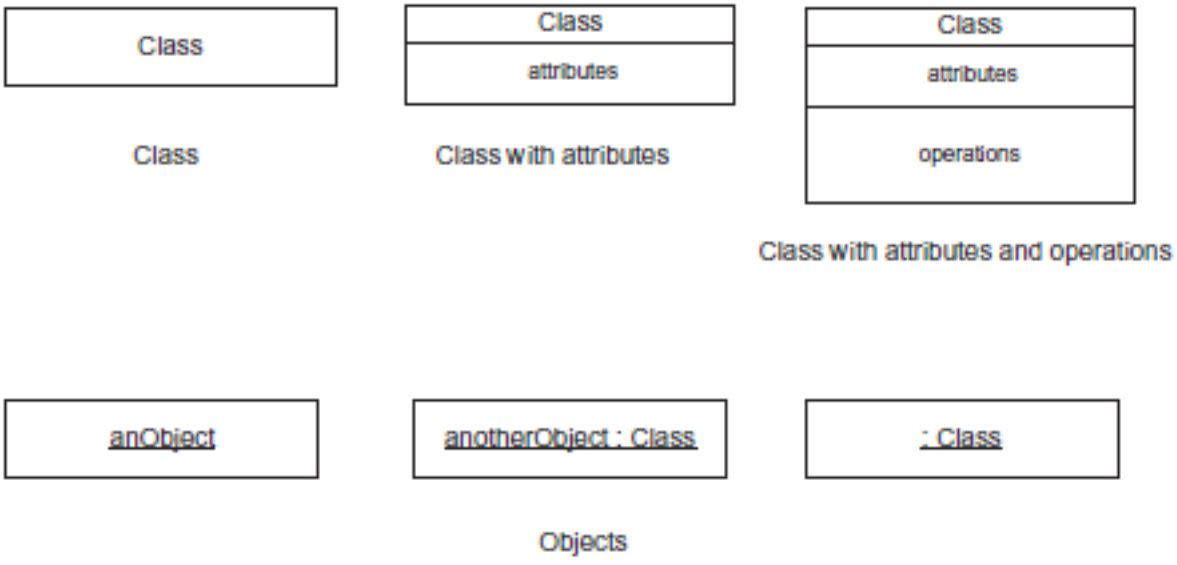
- ✓ **Use Case:** Use Case Name
- ✓ **Short Description:** A Brief Description of the Use Case

- ✓ **Participating Actors**
- ✓ **Pre-Conditions:** A description of the conditions that must be satisfied before the use case is invoked
- ✓ **Post-Conditions:** A description of what has happened at the end of the use case
- ✓ **Main Flow:** A list of the system interactions that take place under the most common scenario. For example, for withdraw money, this would be entering card, enter pin, etc...
- ✓ **Alternate Flow(s):** A description of possible alternative interactions.
- ✓ **Exception Flow(s):** A description of possible scenarios where unexpected or unpredicted events have taken place

2. Class Diagram

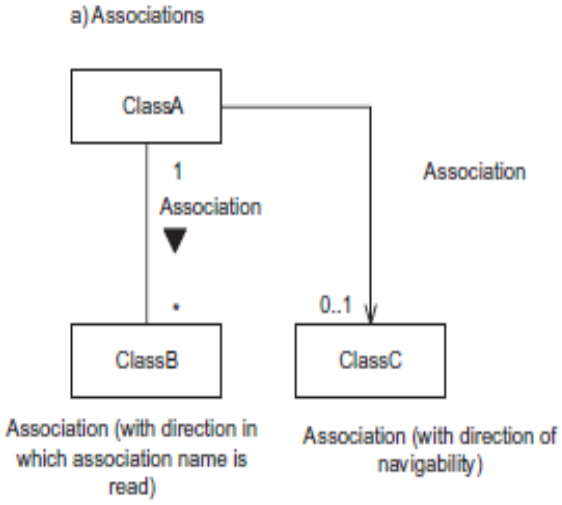
Class diagrams can be derived from use-case diagrams or from text analysis of the given problem domain. Used to define the static structure of classes in a system and their associations. generated by system analysts and designers and will be iteratively refined in the subsequent phases during the software development life cycle.

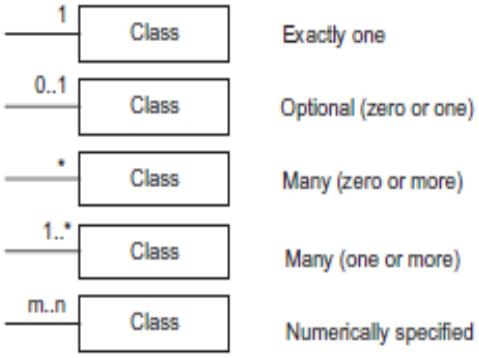
- ✓ The class box always holds the class name.
- ✓ Optionally, the *attributes* and *operations* of a class may also be depicted.
- ✓ To distinguish between a class (the type) and an object (an instance of the type), an object name is shown underlined



❖ **Types of relationships in use Class diagram**

- ✓ Relationships b/n classes can be:
  - Association,
  - Whole/part relationship, and
  - Generalization and specialization.
- ✓ On each end of the association line joining the classes is the multiplicity of the association.
- ✓ The multiplicity of an association specifies how many instances of one class may relate to a single instance.

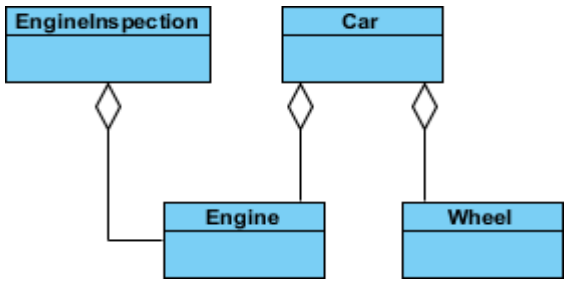




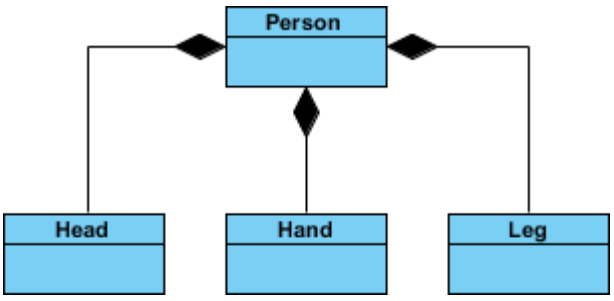
❖ Generalization/Specialization Hierarchies:

- ✓ A generalization/specialization hierarchy is an *inheritance* relationship.
- ✓ Depicted as an arrow joining the *subclass* (child) to the *superclass* (parent),

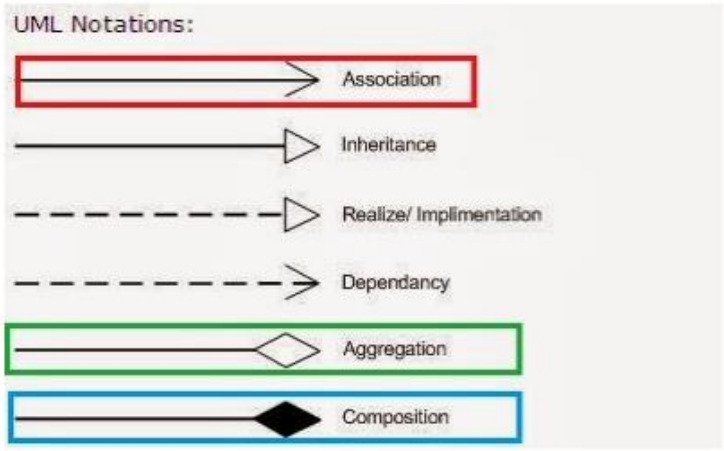
❖ **Aggregation:** In an aggregation relationship, the associated objects exist independently within the scope of the system. Example A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well.

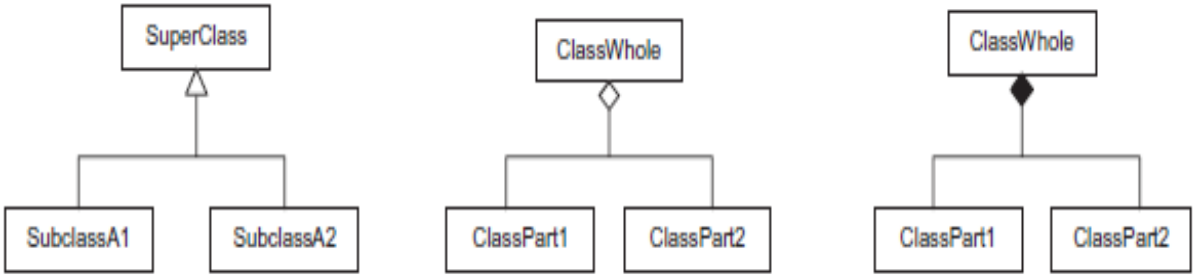


❖ **Composition:** is a strong form of aggregation in which the "whole" is completely responsible for its parts and each "part" object is only associated to the one whole object.



Here are UML notations for a different kind of dependency between the two classes.



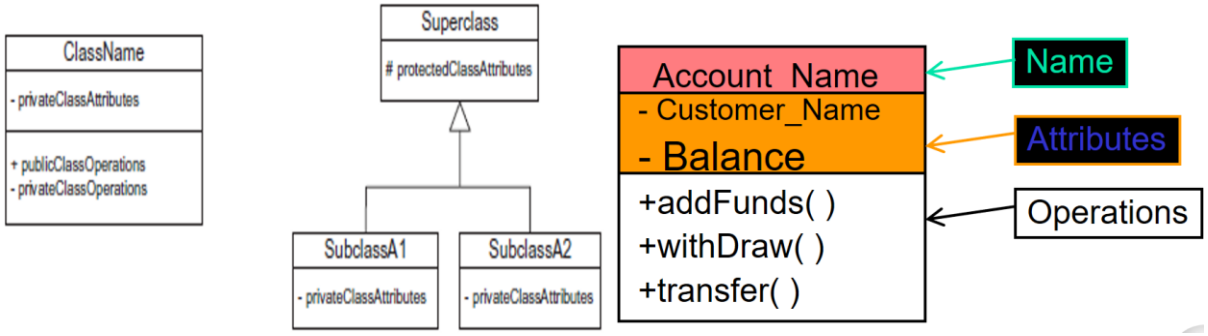


Generalization/specialization Aggregation hierarchy Composition hierarchy

Class diagram Visibility

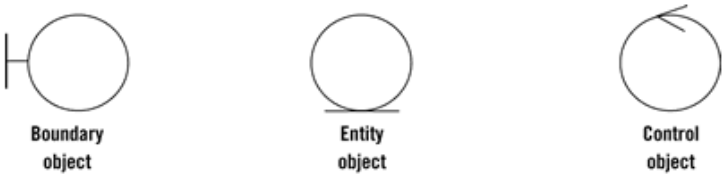
- ✓ Visibility refers to whether an *element* of the class is visible from outside the class.
- ✓ Depicting visibility is *optional* on a class diagram.

- ✓ *Public* visibility, denoted with a + symbol (visible outside)
- ✓ *Private* visibility, denoted with a – symbol (visible only in the class)
- ✓ *Protected* visibility, denoted with a # symbol (class + subclass)



3. Sequence Diagram

It to show sequence of operations or how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. It shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are



Entities *(model)*
Objects representing system data, often from the domain model.

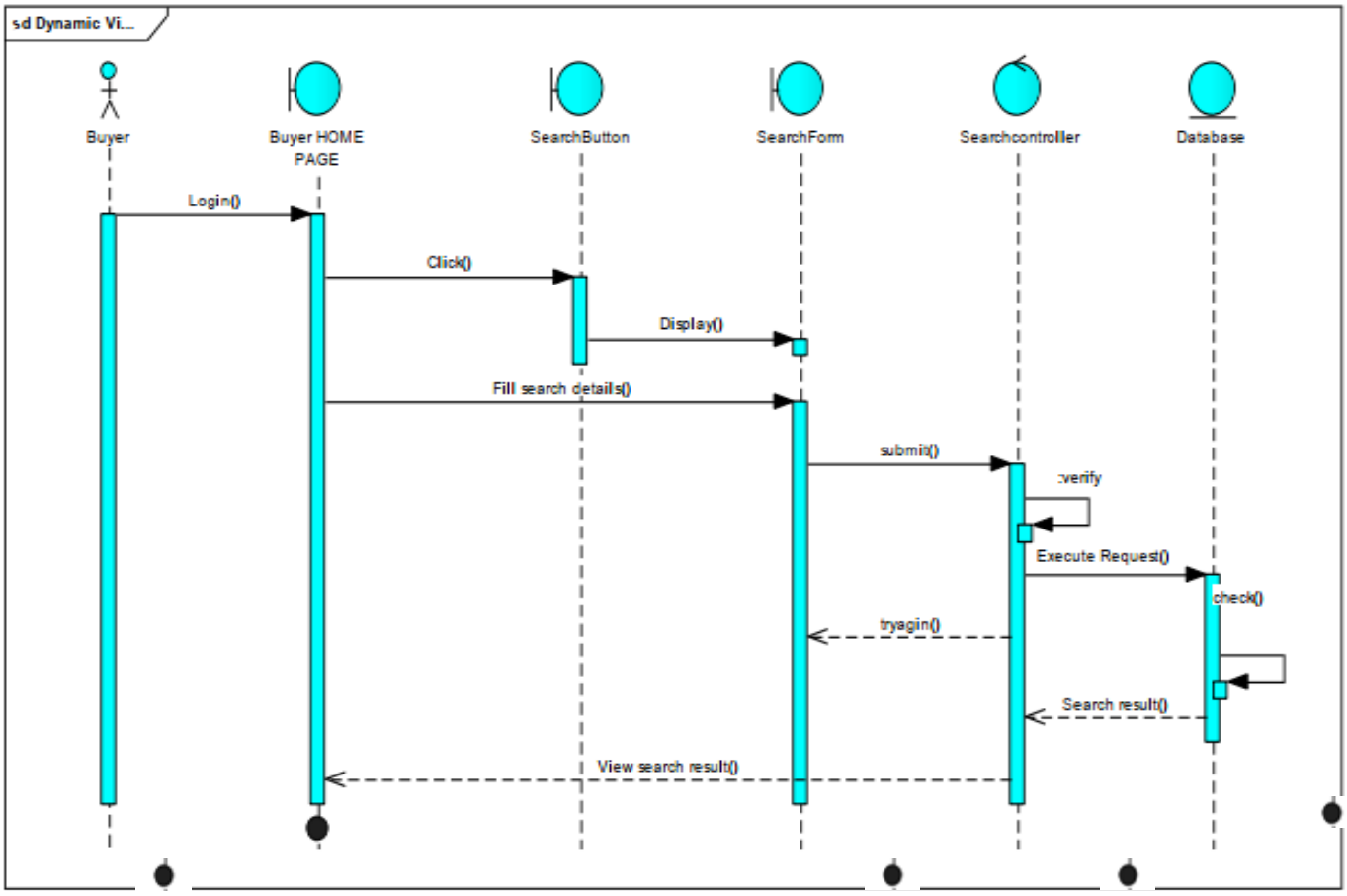
Boundaries *(view/service collaborator)*
Objects that interface with system actors (e.g. a **user** or **external service**). Windows, screens and menus are examples of boundaries that interface with users.

Controls *(controller)*
Objects that mediate between boundaries and entities. These serve as the glue between boundary elements and entity elements, implementing the logic required to manage the various elements and their interactions. It is important to understand that you may decide to implement controllers within your design as something other than objects – many controllers are simple enough to be implemented as a method of an entity or boundary class for example.

Four rules apply to their communication:

1. Actors can only talk to boundary objects.

typically associated with use case realizations in the Logical View of the system under development.



3.6 Characteristics of a Good Requirement

Chapter 4: Project Management

The objective of this chapter is to introduce software project management.

When Student will know the principal tasks of software project managers;

- ✓ *have been introduced to the notion of risk management and some of the risks that can arise in software projects;*
- ✓ *understand factors that influence personal motivation and what these might mean for software project managers;*
- ✓ *Understand key issues that influence team working, such as team composition, organization, and communication.*

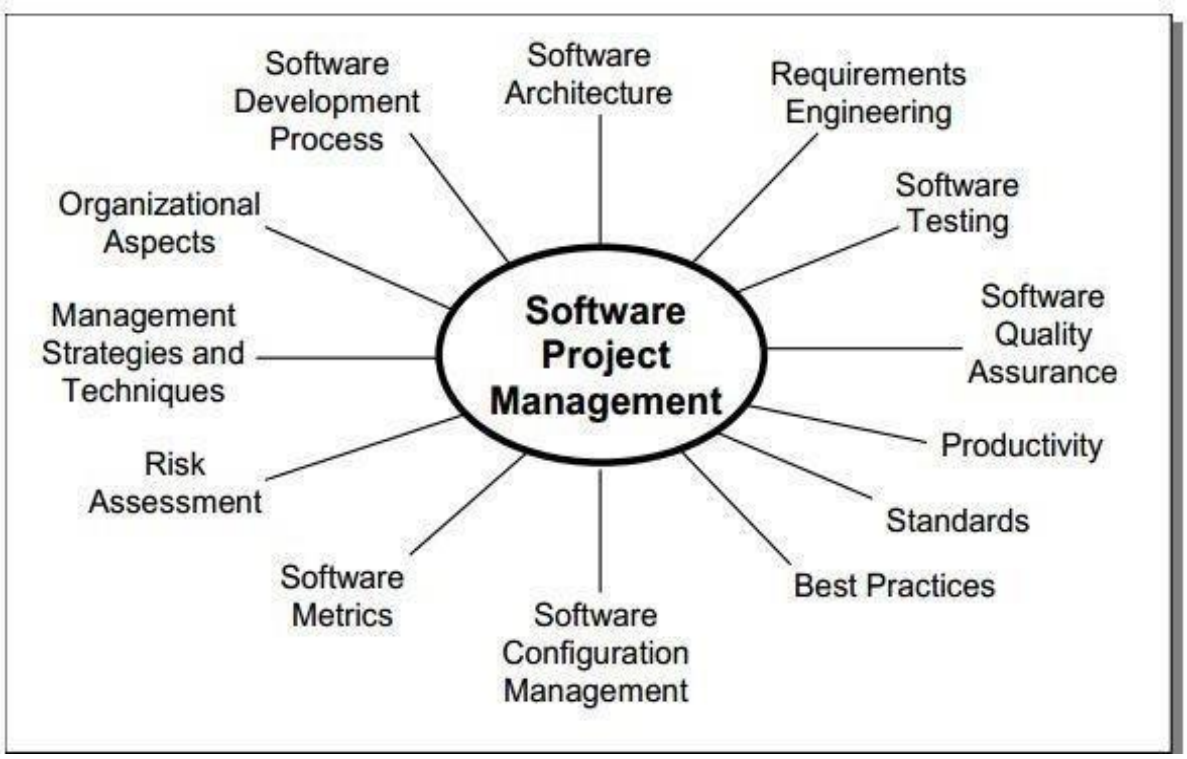
4.1 Project

A project is a temporary endeavor undertaken to create a unique product with defined start and end dates to achieve one or more objectives within the constraints of cost, schedule/time, and quality performance. It is well-defined task, which is a collection of several operations done in order to achieve a goal.

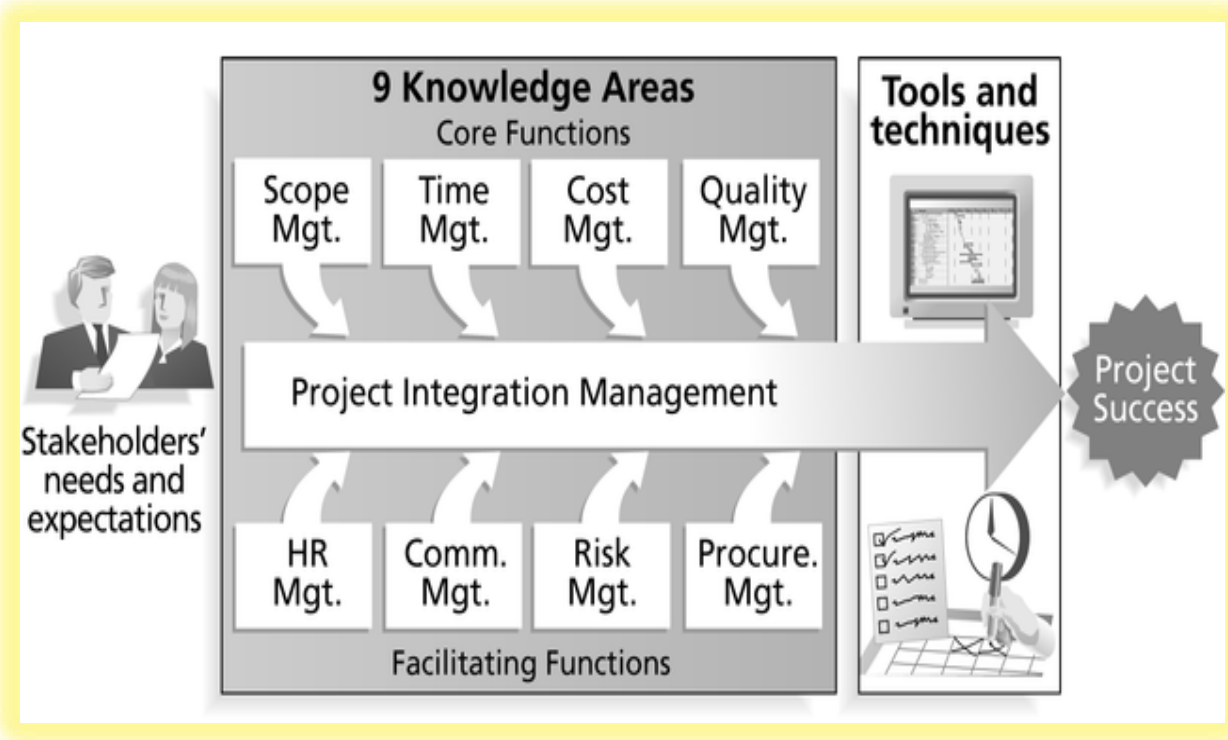
- ✓ Every project may have a unique and distinct goal.
- ✓ Project comes with a start time and end time.
- ✓ Project needs adequate resources in terms of time, manpower, finance, material and knowledge-bank.

4.2 Software Project Management

SPM is a sub-discipline of PM in which software projects are planned, implemented, monitored and controlled. It is an art and science of planning and leading software projects. So, cconcerned with activities involved in ensuring that software is delivered on time/schedule and on budget according to the requirements. Project management is needed because software development is always subject to budget and schedule constraints that are set by the organization.



4.3 Project Management Framework/ Knowledge Areas



Software Project Manager

- ✓ A software project manager is a person who undertakes the responsibility of executing the software project.
- ✓ Project manager may never directly have involved in producing the end product but he controls and manages the activities involved in production.
- ✓ But, closely monitors the development process, prepares various plans, arranges necessary resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

Suggested Skills for a Project Manager

- ✓ Communication skills: listening
- ✓ Organizational skills: planning, analyzing
- ✓ Team Building skills: empathy, motivation
- ✓ Leadership skills: vision, positive
- ✓ Coping skills: flexibility, creativity, patience
- ✓ Technological skills: experience, project knowledge
- ✓ General management skills: like monitoring
- ✓ Conflict resolution, conflict management skills, (negotiating)
- ✓ Critical thinking, problem solving skills and soon

Essentially, 4P's refers to four critically important elements of PM.

Embracing these elements in your project can help your team meet its goals and objectives. This **four P's are:**

1. **People** (people who are play in PM includes project manager, project team members, sponsors, stakeholders, developers.)
2. **Product** (As the name implies, this is **deliverable** of the project)/the software to be built
3. **Process** (appropriate methodology or process model)
4. **Project** (This is where the project manager's roles and responsibilities come into play).

4.4 Activities in Project Management

4.4.1 Project Planning

Project planning is one of the most important jobs of a software project manager. As a manager, you have to break down the work into parts and assign these to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems. The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

The biggest single problem that afflicts software developing is that of underestimating resources required for a project. Developing accurate project plan is essential to gainn understanding of the resources required, and how these should be applied. Project managers are responsible for planning. estimating and scheduling project development and assigning people to tasks/activities.

❖ **Type of Plan**

✓ **Software development plan**

- The central plan, which describes how the system will be developed.
- A project plan is a document used to coordinate all project planning documents (5-20 pages).
- Specifies the order of work to be carried out, resources, responsibilities, and so on.
- Varies from small and relatively informal to large and very formal.
- Developing a project plan is as important as properly designing code:
- On the basis of a project plan, contracts will be signed and careers made or broken.

Its main purpose is:

- To guide project execution.
- To support project manager for leading project team and assessing project status.

✓ **Quality assurance plan.**

- Specifies the quality procedures & standards to be used.

✓ **Validation plan.**

- Defines how the client will validate the system that has been developed.

✓ **Configuration management plan.**

- Defines how the system will be configured and installed.

✓ **Maintenance plan.**

- Defines how the system will be maintained.

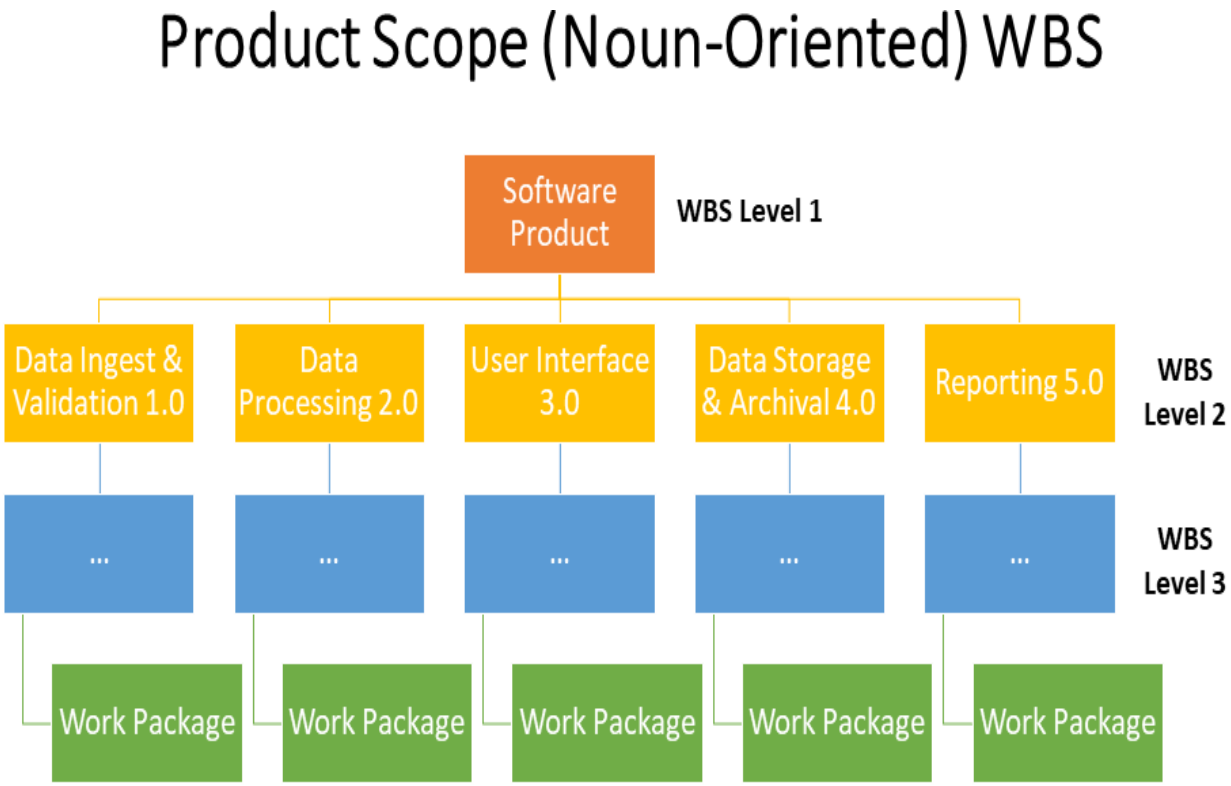
✓ **Staff development plan.**

- Describes how the skills of the participants will be developed.

❖ **Structure of Development Plan**

1. **Introduction**
   - ✓ brief intro to project — references to requirements spec
2. **Project organization**
   - ✓ intro to organizations, people, and their roles
   - ✓ Description of how the project is organized
3. **Risk Analysis**
   - ✓ what are the key risks to the project?
4. **Hardware and software resources**
   - ✓ what h/ware and s/ware resources will be required for the project and when?
5. **Work breakdown**
   - ✓ the project divided into activities, milestones, deliverables; dependencies between tasks etc.

Subdividing major project deliverables and total project scope/ work into smaller, more manageable components. WBS is an outcome-oriented analysis of work involved in a project that defines hierarchical decomposition of total scope of the project. For internal project, work is actually detailed in work breakdown structure (WBS) and WBS dictionary. WBS is a document that break down all the work which needs to be done in the project, then assigns all tasks to the team members. WBS dictionary is a document that provides detailed deliverables, activity and scheduling info about each component in the WBS.



Approaches to Developing WBSs

- ✓ The analogy approach: It often helps to review WBSs of similar projects
- ✓ The top-down approach: Start with the largest items of the project and keep breaking them down.
- ✓ The bottom-up approach: Start with the detailed tasks and roll them up.
- ✓ By using approach above the activities in a project, breakdown into work packages, tasks, deliverables and milestones.

❖ **A work package is a task at the lowest level of the WBS.**

- ✓ a large, logically distinct section of work:
- ✓ typically, at least 12 months' duration;
- ✓ may include multiple concurrent activities;

- ✓ independent of other activities;
- ✓ but may depend on, or feed into other activities;
- ✓ typically allocated to a single team.

❖ **A task is typically a much smaller piece of work**

- ✓ A part of a work package.
- ✓ typically, 3–6 person months' effort;
- ✓ may be dependent on other concurrent activities;
- ✓ typically allocated to a single person.

❖ **A deliverable is an output of the project that can meaningfully be assessed.**

- ✓ A deliverable is a product produced as part of a project, such as hardware or software, planning documents.
  Examples: – report (e.g. requirements spec); code (e.g., alpha tested product).
- ✓ A milestone is a point at which progress on the project may be assessed.
- ✓ Typically, a major turning point in the project
- ✓ EXAMPLES: – delivery of requirements spec delivery of alpha tested code.

6. Project schedule

- ✓ actual time required — allocation of dates

7. Reporting and progress measurement: mechanisms to monitor progress.

4.4.2 Project Scheduling

Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. You estimate the calendar time needed to complete each task, the effort required, and who will work on the tasks that have been identified. You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be. In terms of the planning stages that I discussed in the introduction of this chapter, an initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development

planning.

- Project time management involves the processes required to ensure/manage timely completion of a project.
- It also known as schedule management knowledge area.
- Project scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot assigned to each activities.

Chapter 5: Software Design

At the end of this chapter the students will be able to:

- *Understand meaning software design.*
- *Understand objective of software design.*
- *Understand software design activities.*
- *Identify software design techniques, strategies and approaches*
- *Apply effective software design methodology to design a software system*
- *Design specific and measurable software design to ensure software requirements*

5.1 Software Design

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language. The software design phase is the first step in SDLC**,** which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries. It involves designing the entire system (network, databases, UI, data structures) and its elements, as well as includes overall architecture of the software.

The software design is like a blueprint of the software. This can be related to the construction of a building. Before constructing any building, a blueprint of that building is created and according to that blueprint, further construction is carried out. For a similar purpose, in the case of software development, we make software design. After the design phase, the software undergoes the coding and implementation phase. In this phase, different developers work on different modules. So, for any kind of reference or any help, they refer to this software design. While designing any software, there are two different phases: -

1. Preliminary Design
2. Detailed Design

1. Preliminary Design

In the preliminary design, a rough design of the software is proposed by several designers according to the customer requirements. In the preliminary design, various designs of the software are made, accepted, rejected and modified. After reaching a stage in which the design seems to

fulfil all the customer requirements, a set of trained software designers make the final design of the software.

2. Detailed Design

The final design that is made at the end of the preliminary design phase is called the detailed design. After this design is made, no further changes are made in the design of the software, and the design is sent to the development team for further work. The design phase of the software is very crucial in the software development process. This is because if the design of the software is not accurate to fulfil all the requirements of the user, then the software that will be developed will also look those things because the software will follow the design only. So, for making good quality software, the design of the software should also be proper and accurate. So, each design in the preliminary stage must be evaluated in every manner before being finalized as the design for the software.

5.1.1 Software Design Levels

Software design yields three levels:

1. **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
2. **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
3. **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules. It includes:
    - ✓ Decomposition of major system components into program units.
    - ✓ Allocation of functional responsibilities to units.
    - ✓ User interfaces, Network architecture
    - ✓ Unit states and state changes
    - ✓ Data and control interaction between units

- ✓ Algorithms and data structures

5.2 Objective of Software Design

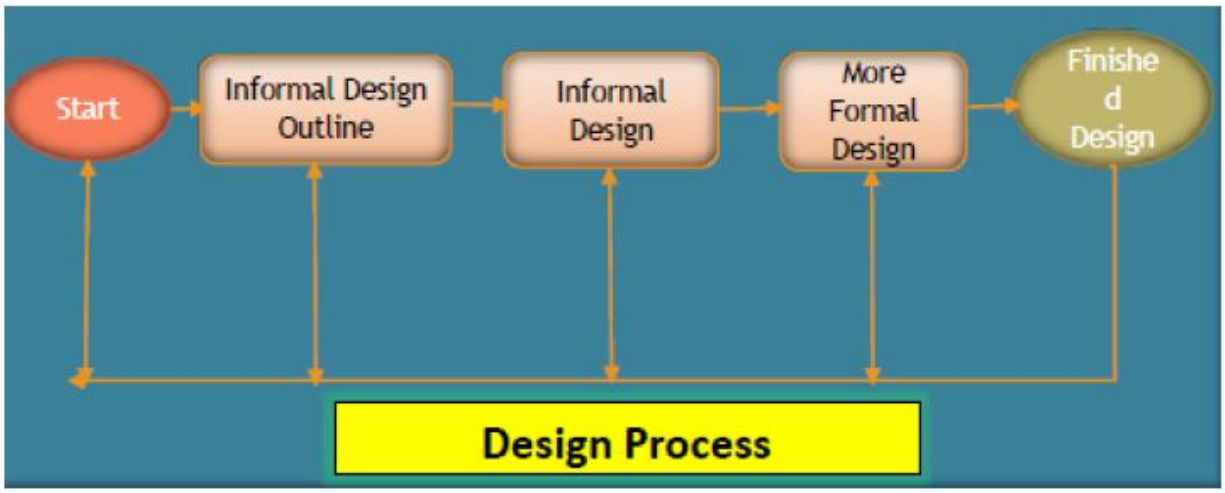
- ✓ Correctness: Software design should be correct as per requirement.
- ✓ Completeness: The design should have all components like data structures, modules, and external interfaces, etc.
- ✓ Efficiency: Resources should be used efficiently by the program.
- ✓ Flexibility: Able to modify on changing needs.
- ✓ Consistency: There should not be any inconsistency in the design.
- ✓ Maintainability: The design should be so simple so that it can be easily maintainable by other designers

5.3 Stages of Design

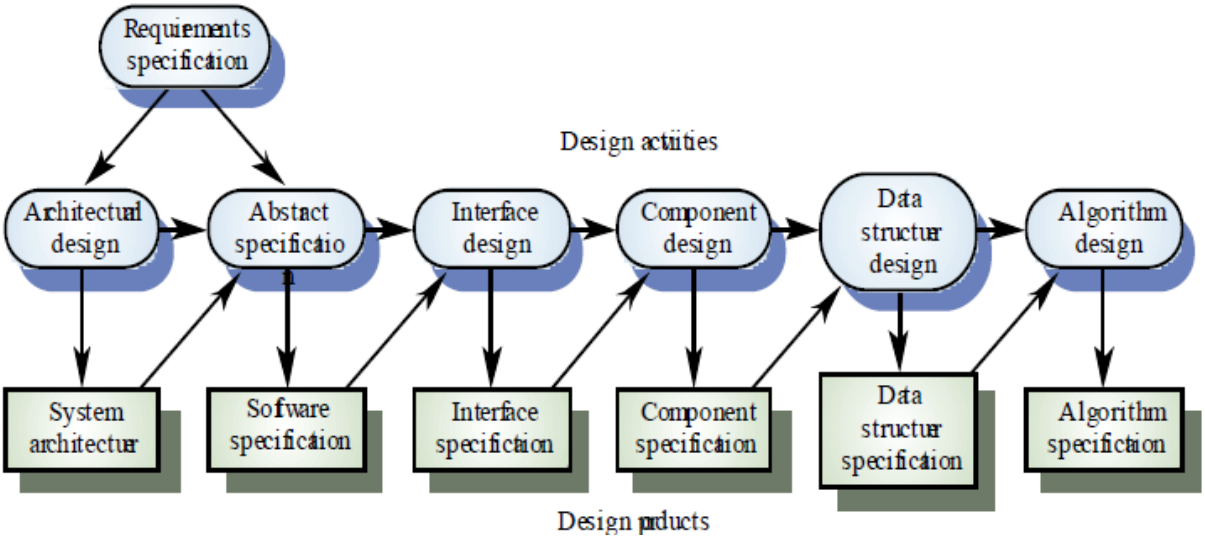
- ✓ Problem Understanding
  - ✦ Look at the problem from different angles to discover the design requirements.
- ✓ Identify one or more solutions
  - ✦ Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
  - ✦ Describe solution abstractions
  - ✦ Use graphical, formal or other descriptive notations to describe the components of the design.
- ✓ Repeat the process for each identified abstraction until the design is expressed in primitive terms.

5.4 Design Process

Software designers do not arrive at a finished design immediately. So, they develop design iteratively through a number of different versions. starting point is informal design which is refined by adding information to make it consistent and complete design.



5.4.1 Software Design Activities



It includes the following software design activities:

- ✓ Architectural Design: Identify sub-systems/layouts.
- ✓ Abstract Specification: Specify sub-systems.
- ✓ Interface Design: Describe sub-system interfaces interaction.
- ✓ Component Design: Decompose sub-systems into components.
- ✓ Data Structure Design: Design data structures to hold problem data.
- ✓ Algorithm Design: Design algorithms for problem functions.

5.4.2 Software Design Activities

The fundamental activity that are common to all software process includes 7 steps in software design process activities: -

Step1: Determining the issues

Step2: Conducting extensive research

Step3: Coming up with a possible solution

Step4: Assessing and identifying a real solution

Step5: Developing and evaluating

Step6: Debugging and evaluating

Step7: Make changes to the finished product

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.

A. 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.

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.

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.

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.

B. 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 analyse 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.

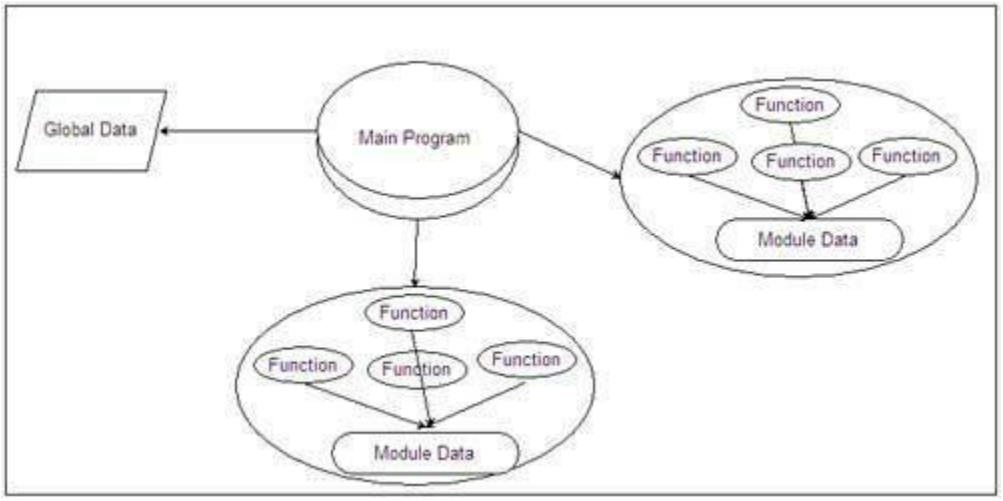
C. Patterns

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

- ✓ Whether the pattern can be reused
- ✓ Whether the pattern is applicable to the current project
- ✓ Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

D. Modularity

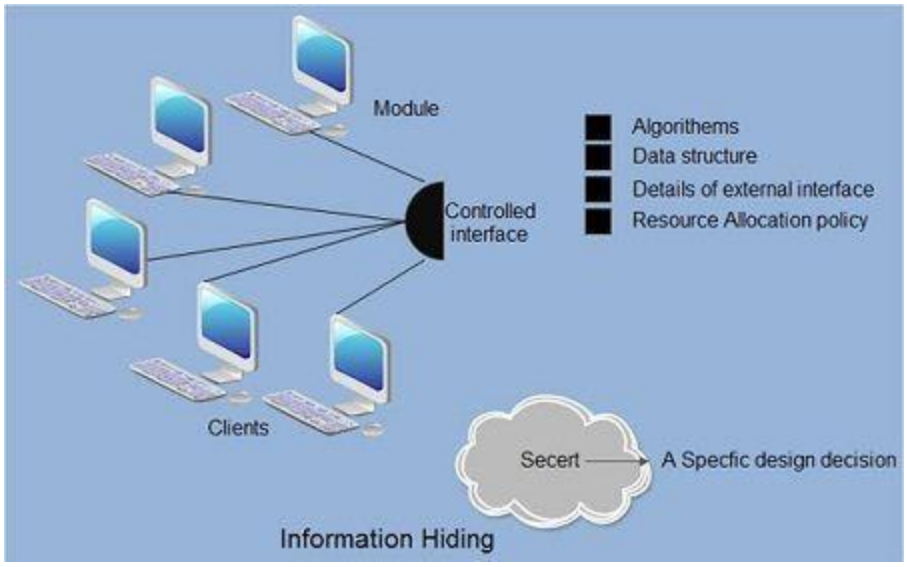
Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing 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.

E. 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.

- ✓ Leads to low coupling
- ✓ Emphasizes communication through controlled interfaces
- ✓ Decreases the probability of adverse effects
- ✓ Restricts the effects of changes in one component on others
- ✓ Results in higher quality software.

F. 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.

INPUT

Get user's name (string) through a prompt.

Get user's grade (integer from 0 to 100) through a prompt and validate.

PROCESS

OUTPUT

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

INPUT

Get user's name through a prompt.

Get user's grade through a prompt.

While (invalid grade)
Ask again:
PROCESS

OUTPUT

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

G. 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. Refactoring a software means examining the existing software design for:

- ✓ Redundancy
- ✓ Unused design elements
- ✓ Inefficient or unnecessary algorithms
- ✓ Poorly constructed or inappropriate data structures
- ✓ And any other design failure, in order to improve design or to get a better design.

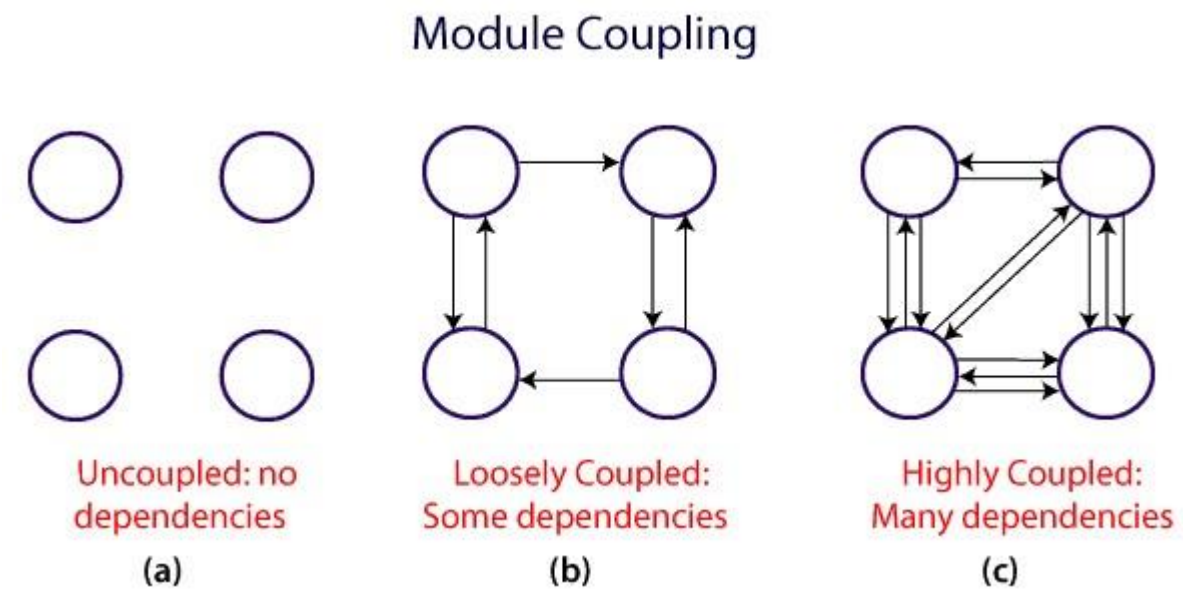
5.5 Functional independence (Coupling and Cohesion)

Module Coupling

In software engineering, coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. Uncoupled modules have no interdependence at all within them. Two components can be dependent in many ways:

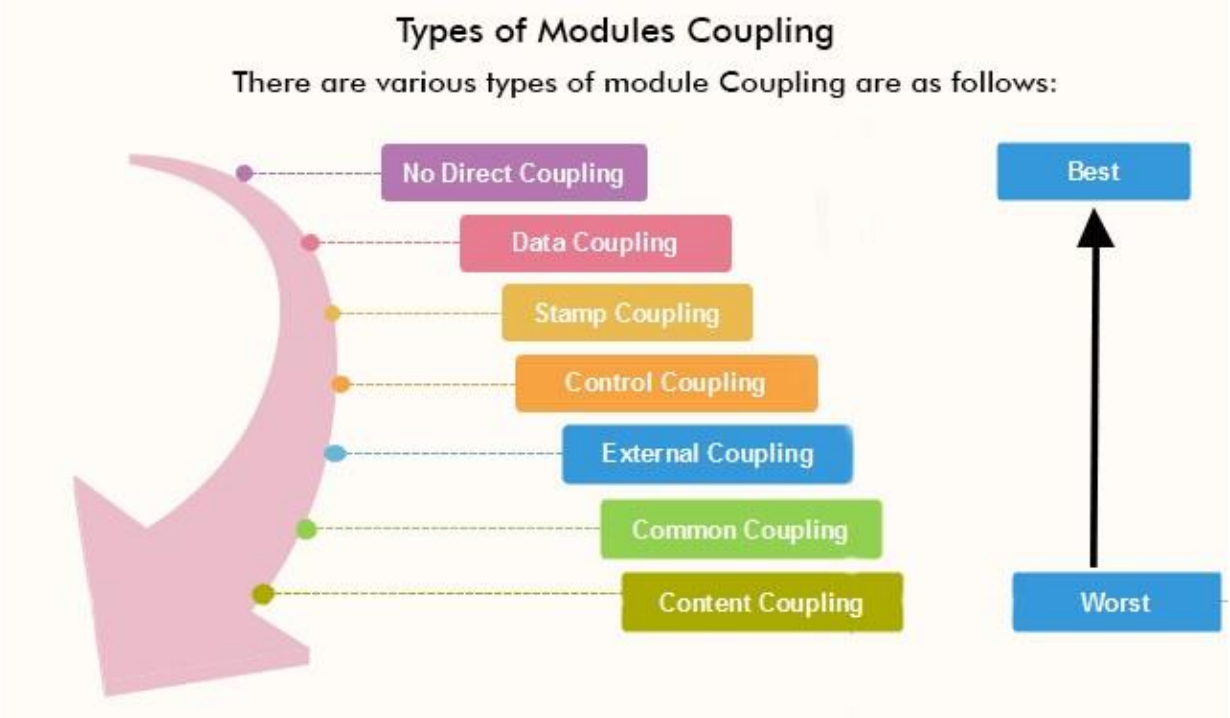
- ✓ References made from one to another
- ✓ Amount of data passed from one to another
- ✓ Amount of control one has over the other
- ✓ The complexity of the interface
- ✓ The type of information flow between modules

The various types of coupling techniques are shown in fig:

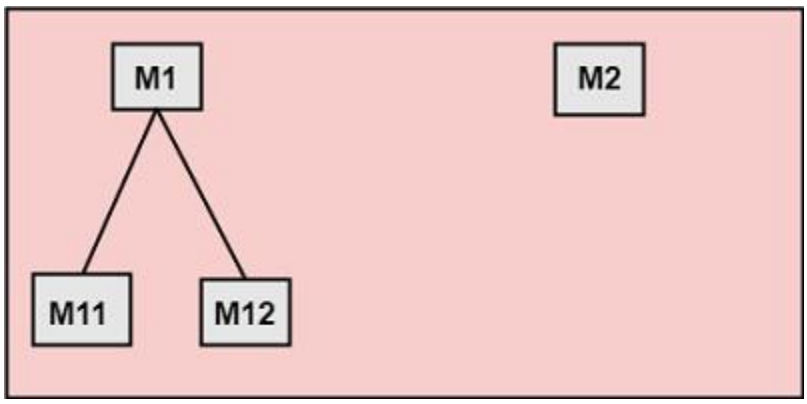


A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

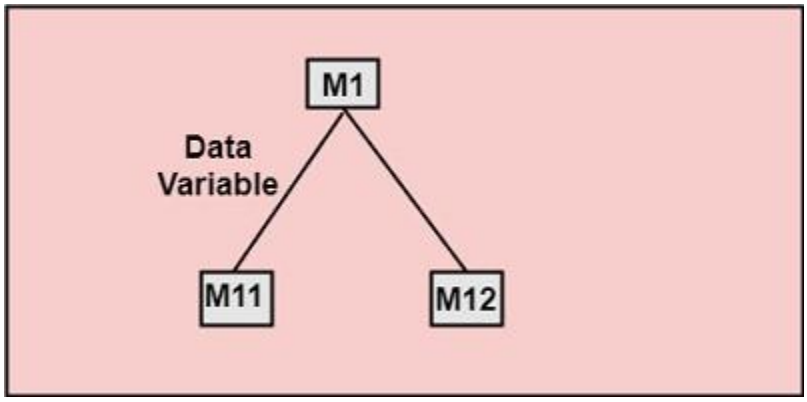


1. No Direct Coupling: There is no direct coupling between M1 and M2.

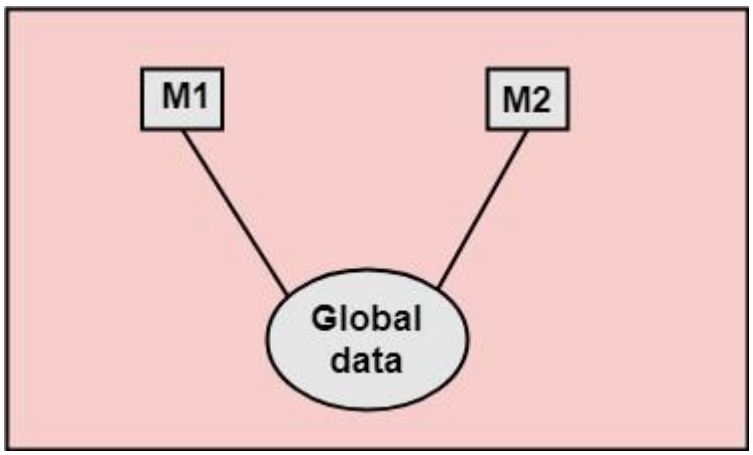


In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. **Data Coupling:** When data of one module is passed to another module, this is called data coupling.



3. **Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.
4. **Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.
5. **External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.
6. **Common Coupling:** Two modules are common coupled if they share information through some global data items.

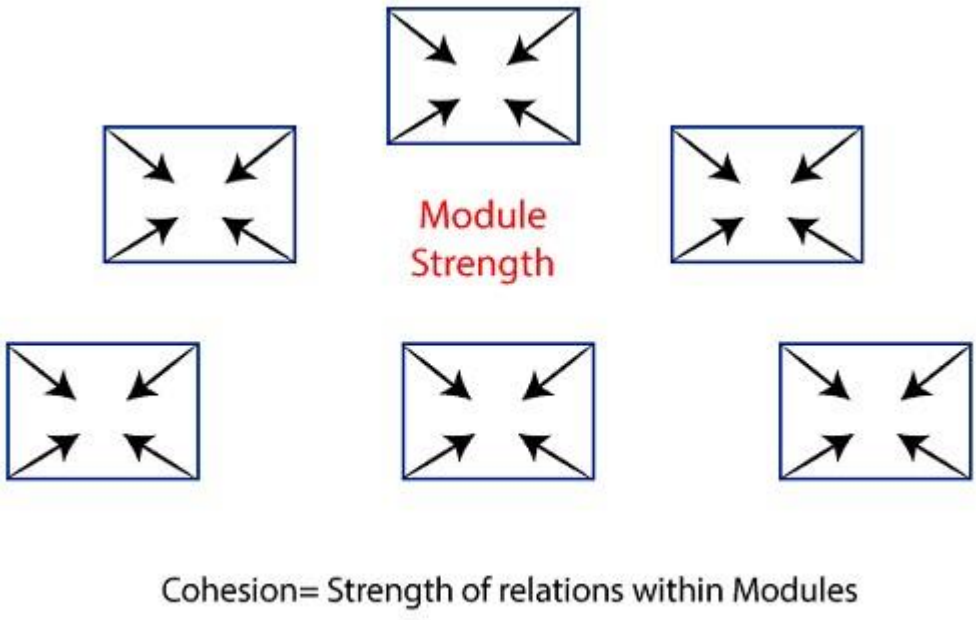


7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

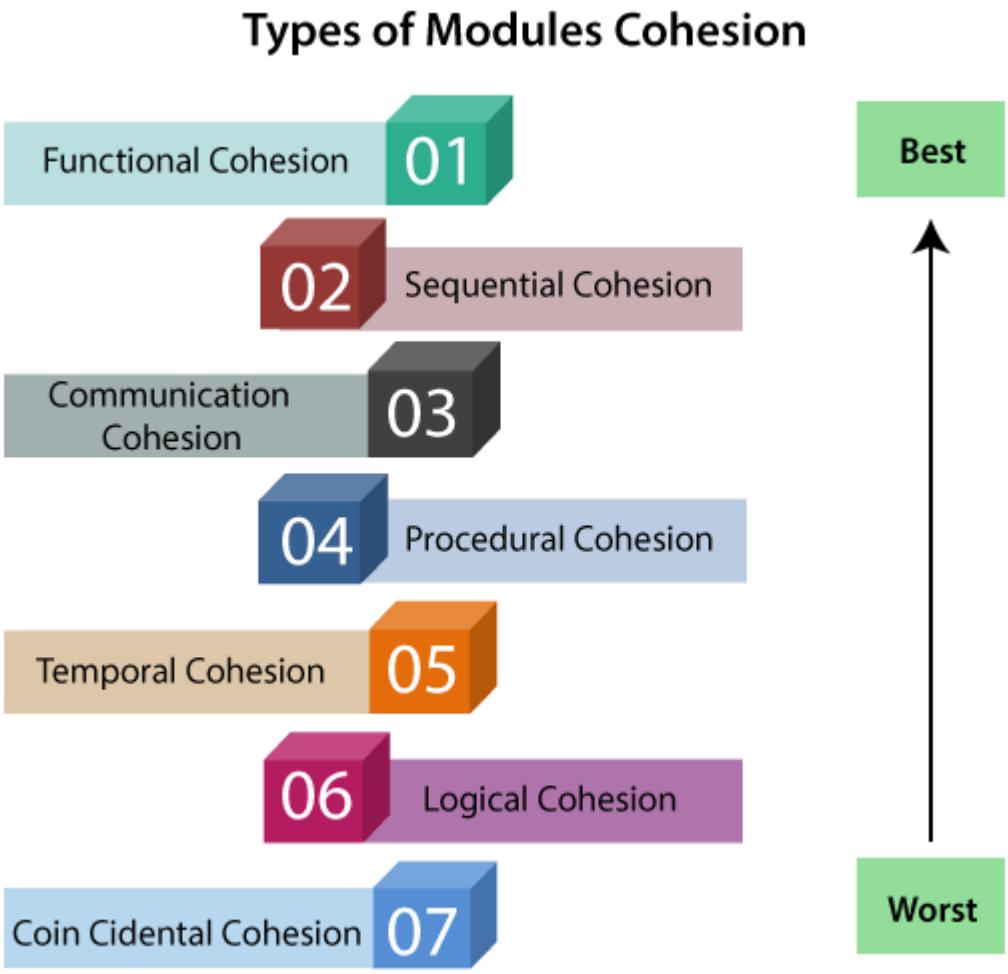
Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an ordinal type of measurement and is generally described as "high cohesion" or "low cohesion."



Types of Modules Cohesion



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module forms the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example, Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

5.6 Software Design Strategies/ Methodology

5. Function Oriented Design

Function oriented design is the result of focusing attention to the function of the program. This is based on the stepwise refinement. Stepwise refinement is based on the iterative procedural decomposition. Stepwise refinement is a top-down strategy where a program is refined as a hierarchy of increasing levels of details.

We start with a high level description of what the program does. Then, in each step, we take one part of our high level description and refine it. Refinement is actually a process of elaboration. The process should proceed from a highly conceptual model to lower level details. The refinement of each module is done until we reach the statement level of our programming language.

Characteristic of Function oriented design

- ✓ inherits some properties of structured design where divide and conquer methodology is used.
- ✓ This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.
- ✓ These functional modules can share information among themselves by means of information passing and using information available globally.
- ✓ when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules.
- ✓ **Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.**

6. Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system.it is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, it is orthogonal to function -oriented design. Object-oriented design begins with an examination of the real world “things”. These things are characteristics individually in terms of their attributes and behavior.

Objects are independent entities that may readily be changed because all state and representation information is held within the object itself. Object may be distributed and may execute sequentially or in parallel. Object oriented technology contains following concept:

5. Objects

Software package are designed and developed to correspond with real world entities that contain all the data and services to function as their associated entities messages. All entities involved in the solution design are known as objects.

- ✓ For example, person, banks, company and customers are treated as objects.
- ✓ Every entity has some attributes associated to it and has some methods to perform on the attributes.

6. Classes

A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

7. Encapsulation

In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world.

❖ **Inheritance**

OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

❖ **Polymorphism**

OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

5.6.1 Object Oriented Design System design activities: From objects to subsystems

1. Identify design goals from the non-functional requirements
2. Design an initial subsystem decomposition
3. Map subsystems to processors and components
4. Decide storage
5. Define access control policies
6. Select a control flow mechanism
7. Identify boundary conditions

1. Identify design goals from the non-functional requirements

Is the first step of system design, it identifies the qualities that our system should focus on; which are mostly inferred from the non-functional requirements or from the application domain.

Design goals for MyTrip

- ✓ **Reliability:** MyTrip should be rreliable [generalization of non-functional requirement 2].
- ✓ **Fault Tolerance:** MyTrip should be fault tolerant to loss of connectivity with the routing service [rephrased non-functional requirement 2].

- ✓ **Security:** MyTrip should be secure, i.e., not allow other drivers or no authorized users to access another driver's trips [deduced from application domain].
- ✓ **Modifiability:** MyTrip should be modifiable to use different routing services [**anticipation of change by developers**].

list a number of possible design criteria.

- ✓ **Performance: include the speed and space requirements imposed on the system.**

Design criterion	Definition
Response time	How soon is a user request acknowledged after the request has been issued?
Throughput	How many tasks can the system accomplish in a fixed period of time?
Memory	How much space is required for the system to run?

- ✓ **Dependability:** Determine how much effort should be expended in minimizing system crashes and their consequences. How often can the system crash? How available to the user should the system be? Are there safety issues associated with system crashes? Are there security risks associated with the system environment?

Design criterion	Definition
Robustness	Ability to survive invalid user input
Reliability	Difference between specified and observed behavior.
Availability	Percentage of time system can be used to accomplish normal tasks.
Fault tolerance	Ability to operate under erroneous conditions.
Security	Ability to withstand malicious attacks
Safety	Ability to not endanger human lives, even in the presence of errors and failures.

- ✓ **Cost: include the cost to develop the system, to deploy it, and to administer it.**

Design criterion	Definition
Development cost	Cost of developing the initial system
Deployment cost	Cost of installing install the system and training the users.
Upgrade cost	Cost of translating data from the previous system. This criteria results in backward compatibility requirements.
Maintenance cost	Cost required for bug fixes and enhancements to the system
Administration cost	Money required to administer the system.

- ✓ **Maintenance**

Determine how difficult it is to change the system after deployment. How easily can new functionality be added? How easily can existing functions be revised? Can the system be adapted to a different application domain? How much effort will be required to port the system to a different platform?

Design criterion	Definition
Extensibility	How easy is it to add the functionality or new classes of the system?
Modifiability	How easy is it to change the functionality of the system?
Adaptability	How easy is it to port the system to different application domains?
Portability	How easy is it to port the system to different platforms?
Readability	How easy is it to understand the system from reading the code?
Traceability of requirements	How easy is it to map the code to specific requirements?

- ✓ **End user criteria:** include qualities that are desirable from a users' point of view that have not yet been covered under the performance and dependability criteria.

Design criterion	Definition
Utility	How well does the system support the work of the user?
Usability	How easy is it for the user to use the system?

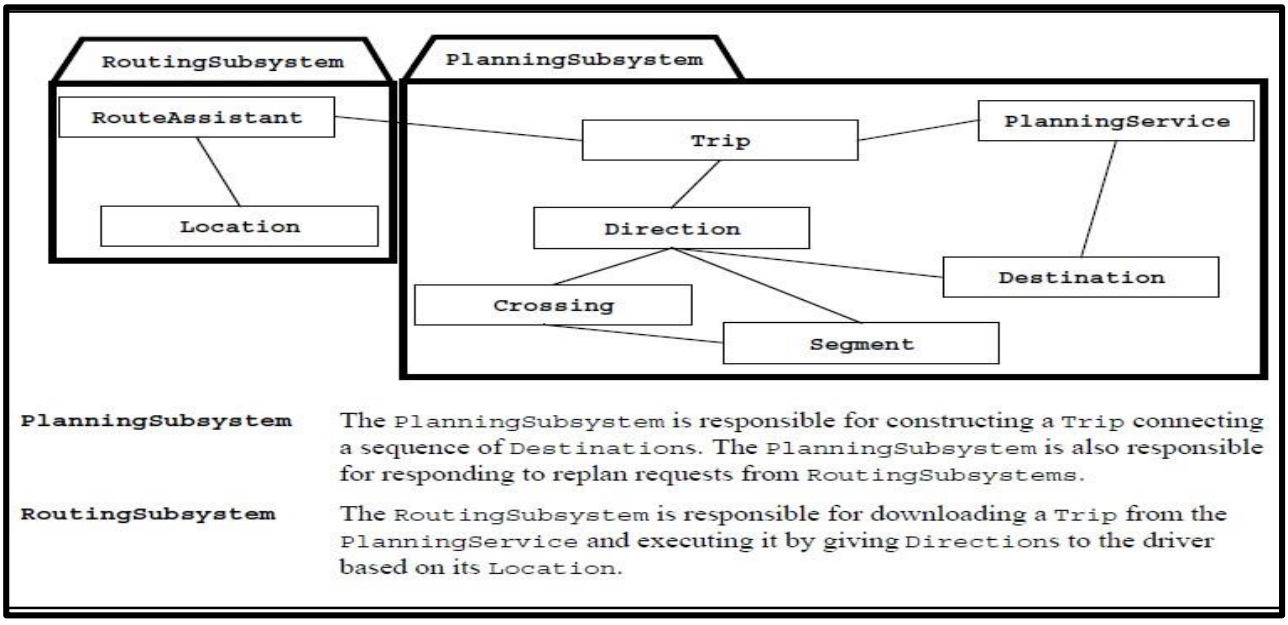
2. Design an initial subsystem decomposition

Finding subsystems during system design has many similarities to finding objects during analysis: It is a volatile activity driven by heuristics. The initial subsystem decomposition should be derived from the functional requirements.

For example: in the MyTrip system, we identify two major groups of objects:

1. PlanTrip use cases
2. ExecuteTrip use case.

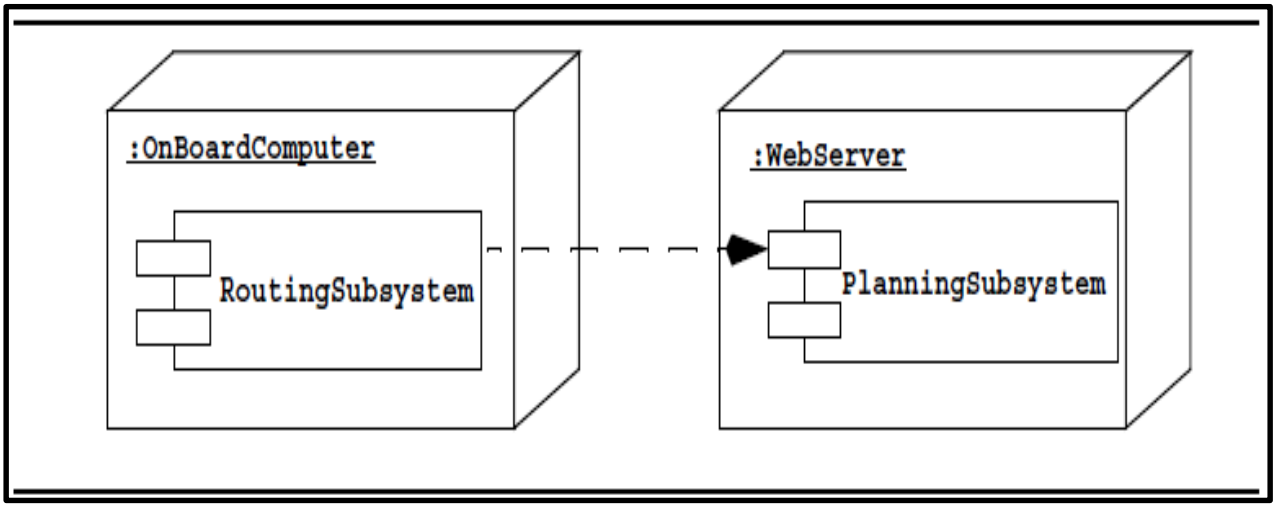
N.B. The Trip, Direction, Crossing, Segment, and Destination classes are shared between both use cases.



3. Map subsystems to processors and components

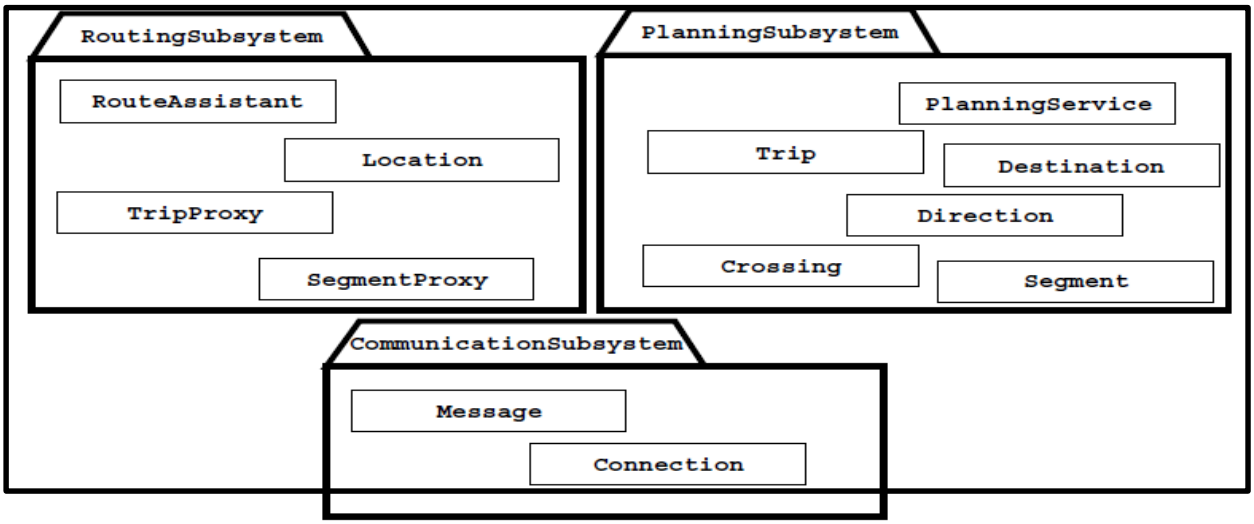
Selecting a hardware configuration and a platform

- ✓ Many web based systems need multiple computers and interconnect multiple distributed users.
- ✓ These computers are modelled as nodes in UML deployment diagrams.
- ✓ Select a hardware configuration also includes selecting a virtual machine onto which the system should be built. It includes OS, s/w (DBMS, communication packages).
- ✓ Example in MyTrip, we deduce from the requirements that Planning Subsystem and Routing Subsystem run on two different nodes: In the MyTrip subsystem We select a Unix machine as the virtual machine for the: WebServer and the Web browsers Netscape and Internet Explorer as the virtual machines for the: OnBoardComputer**.**



Allocating objects and subsystems to nodes

- ✓ After identification of virtual machines, objects and subsystems are assigned to nodes. This often triggers the identification of new objects and subsystems for transporting data among the nodes. Example: in MyTrip system both RoutingSubsystem and PlanningSubsystem share the objects. thus we create new subsystem to support this communication: communication Subsystem.



4. Defining Persistence Data storage

Where and how data is stored in the system impacts the system decomposition. The selection of a specific database management system can also have implications on the overall control strategy and concurrency management. we first need to identify which objects need to be persistent. The persistency of objects is directly inferred from the application domain. **E.g. In Mytrip - current trip in Rroutngsubsystem-file, location X entire trip in PlanningSubsystem-database**

5. **Define Access Control Policies:** In multiuser systems, different actors have access to different functionality and data. We modelled these distinctions by associating different use cases to different actors. During system design, we model access by examining the object model, by determining which objects are shared among actors, and by defining how actors can control access. Depending on the security requirements on the system, we also define how actors are authenticated to the system (i.e., how actors prove to the system who they are) and how selected data in the system should be encrypted.

CommunicationSubsystem	The CommunicationSubsystem is responsible for transporting Trips from the PlanningSubsystem to the RoutingSubsystem. *The CommunicationSubsystem uses the Driver associated with the Trip being transported for selecting a key and encrypting the communication traffic.*
PlanningSubsystem	The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem. *Prior to processing any requests, the PlanningSubsystem authenticates the Driver from the RoutingSubsystem. The authenticated Driver is used to determine which Trips can be sent to the corresponding RoutingSubsystem.*
Driver	*A Driver represents an authenticated user. It is used by the CommunicationSubsystem to remember keys associated with a user and by the PlanningSubsystem to associate Trips with users.*

6. Select a control flow mechanism

Global control describes how the global software control is implemented. In particular, this section should describe how requests are initiated and how subsystems synchronize.

- ✓ Control flow: is the sequencing of actions in a system.
- ✓ In an object-oriented system, sequencing actions includes deciding which operations should be executed and in which order.
- ✓ These decisions are based on external events generated by an actor or on the passage of time.

7. Identify boundary conditions

It is deciding how the system is started, initialized, and shut down, and how to deal with major failures, such as data corruption, whether they are caused by a software error or a power outage. Considering the MyTrip system our system design doden't address the following questions:

- ✓ How MyTrip is initialized and How are maps loaded into the PlanningService?
- ✓ How is MyTrip installed in the car?

- ✓ How does MyTrip know which PlanningService to connect to?
- ✓ How are drivers added to the Planning Service?
- ✓ We discover a set of **use cases** that has not been specified. We call these the **system** administration use cases**,** which specify the system during startup and shutdown phases.

5.6.2 Software Design Approaches

❖ **Top Down Design**

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved. Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence. Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

❖ **Bottom-up Design**

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased. Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

- ✓ Both, top-down and bottom-up approaches are not practical individually.
- ✓ Instead, a good combination of both is used.

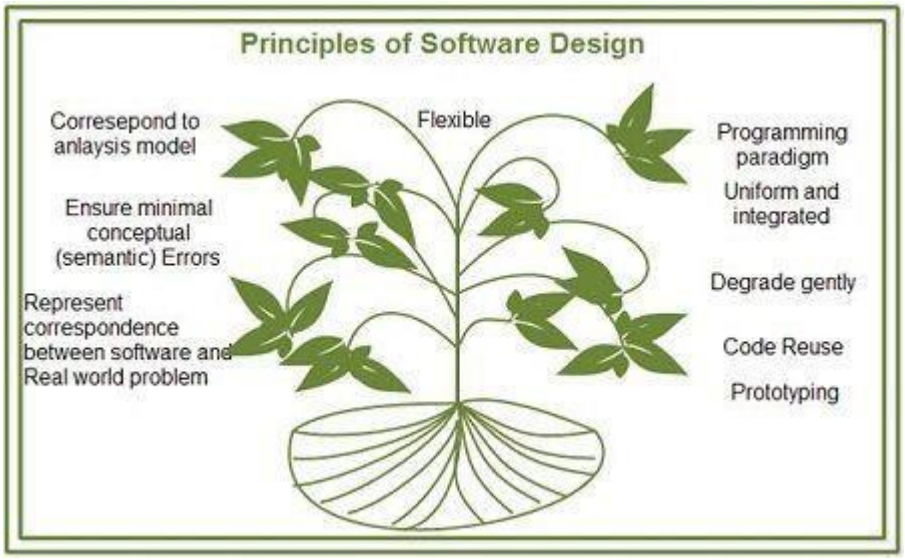
Difference Between Function Oriented Design and Object Oriented Design

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real world functions but are the data

		abstraction where the real world entities are represented.
Function	Functions are grouped together by which a higher level function is obtained.	Function are grouped together on the basis of the data they operate since the classes are associated with their methods.
execute	carried out using structured analysis and structured design i.e., data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function oriented design we decompose in function/procedure level.	We decompose in class level.
Use	This approach is mainly used for computation sensitive application.	This approach is mainly used for evolving system which mimics a business or business case.

5.7 Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.



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 a way 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. **SOLID Principles of Software Architecture:**

Each character of the word SOLID defines one principle of software architecture. This SOLID principle is followed to avoid product strategy mistakes. A software architecture must adhere to SOLID principle to avoid any architectural or developmental failure.

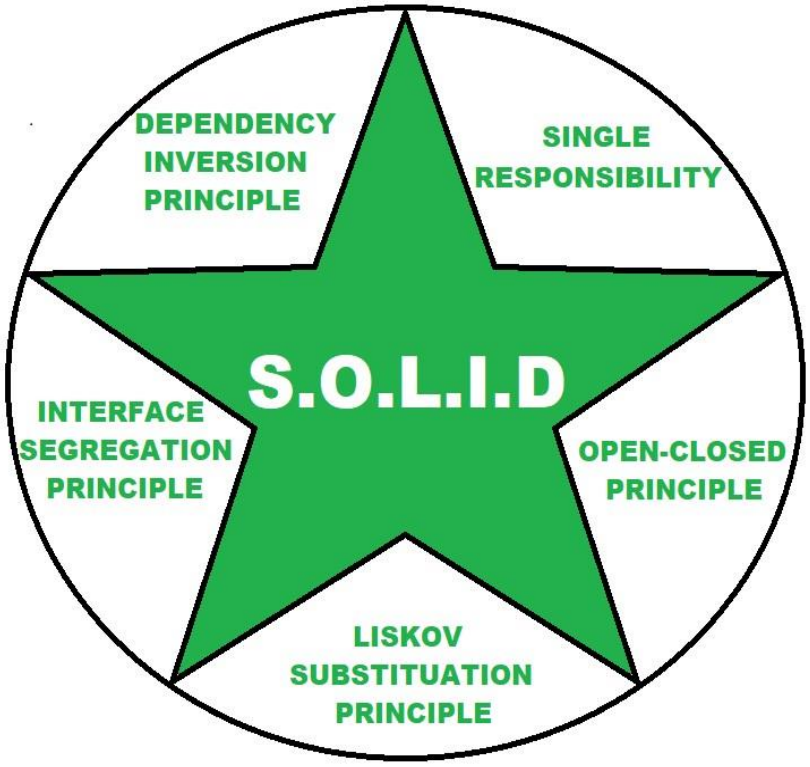


Fig: S.O.L.I.D Principle

1. **Single Responsibility:** Each service should have a single objective.
2. **Open-Closed Principle:** Software modules should be independent and expandable.
3. **Liskov Substitution Principle:** Independent services should be able to communicate and substitute each other.
4. **Interface Segregation Principle:** Software should be divided into such microservices there should not be any redundancies.
5. **Dependency Inversion Principle:** Higher-levels modules should not be depending on low-lower-level modules and changes in higher level will not affect to lower level.

5.8 Architecture Style

Software architecture patterns are related to software architecture styles, but they are unique concepts. Architecture patterns are essentially organizational software templates in which software elements can be placed, while architecture styles function like different formats for the different elements. Software architecture styles are set of rules, constraints, or patterns of how to structure a system into a set of components and connectors.

According to Garlan and Shaw, architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types and a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints-say, having to do with execution semantics—might also be part of the style definition.

An architectural style/ pattern is a set of principles. You can think of it as a coarse-grained pattern that provides an abstract framework for a family of systems. It is a general, reusable solution to a commonly occurring problem in software architecture within a given context. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. Architectural patterns are similar to software design pattern but have a broader scope.

Key architectural styles in Software Engineering are:

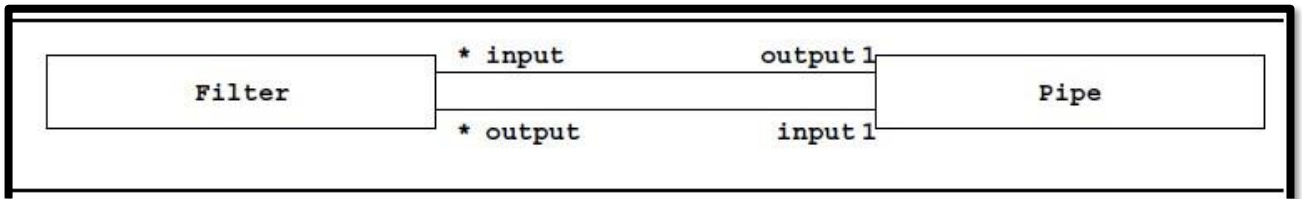
1. Pipes-and-Filter
2. Client-Server
3. Peer-to-Peer
4. Event Based
5. Layering
6. MVC (Model-View-Controller Pattern)
7. SOA (Service Oriented Architecture)

5.8.1 Pipe-Filter

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a filter component. In this architecture subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs. The

subsystems are called filters, and the associations between the subsystems are called pipes. These pipes can be used for buffering or for synchronization purposes.

Data is processed by being separated or filtered between different elements or pipelines, which conjoin their results into an output viewed by the software user, significantly reducing the power necessary to complete the computing task. In other words, data from multiple sources are put into discrete sections of a software program that all concurrently perform the same pattern of tasks on each data set. The end result is multiple datasets, all in the same format, that have been produced using the same tasks. These sets are then supplied to the user and can be analysed, making it a fairly reliable software pattern, but it is also very complex. There is also a small risk for data redundancy. A filter can have many inputs and outputs. A Pipe connects one of the outputs of a filter to one of the inputs of another filter as shown in below:



"The **Pipes and Filters** architectural pattern provides a structure for systems that **process a stream of data**. Each processing step is encapsulated in a filter component. Data [are] passed through pipes between adjacent filters. Recombining filters allows you to build families of related filters."

Implementation steps:

- Divide the functionality of the problem into a sequence of processing steps.
- Define the type and format of the data to be passed along each pipe. ○ Determine how to implement each pipe connection.
- Design and implement the filters.

Note: The design of a filter is based on the nature of the task to be performed and the nature of the pipes to which it can be connected.

- Drawbacks
  - ✦ Encourages batch processing

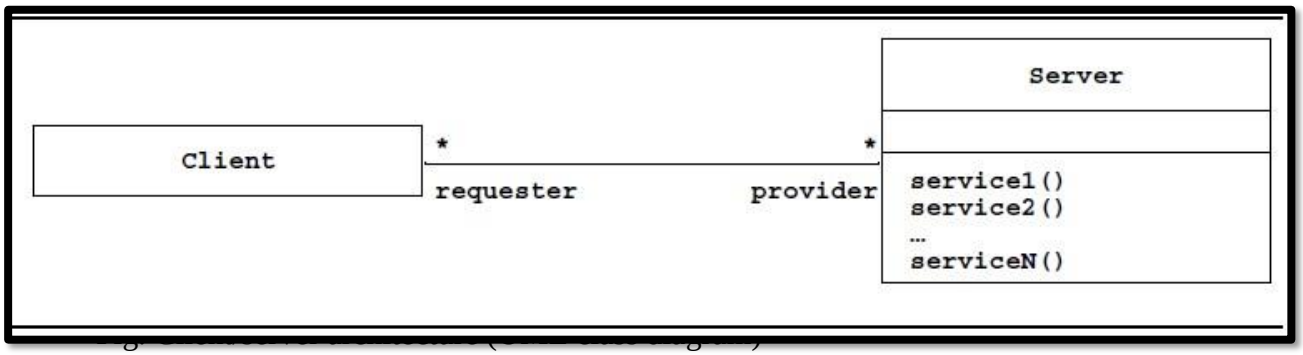
- ✦ Not good for handling interactive application ✦ Duplication in filters functions.

5.8.2 Client-Server

The client-server architecture refers to a system that hosts, delivers, and manages most of the resources and services that the client requests. In this model, all requests and services are delivered over a network, and it is also referred to as the networking computing model or client server network. Client-server architecture, alternatively called a client-server model, is a network application that breaks down tasks and workloads between clients and servers that reside on the same system or are linked by a computer network.

Client-server architecture typically features multiple users' workstations, PCs, or other devices, connected to a central server via an Internet connection or other network. The client sends a request for data, and the server accepts and accommodates the request, sending the data packets back to the user who needs them. A client-server architecture distributes application logic and services respectively to a number of client and server sub systems, each potentially running on a different machine and communicating through the network (e.g, by RPC).

- ✓ Two types of components:
  - ✦ Server components offer services (Response/Reply)
  - ✦ Clients sends request and receive response from server.
- ✓ Client may send the server a request and server reply a response to the request. It is a kind of request and response/reply mechanism.
- ✓ In the client/server architecture a subsystem, the **server,** provides services to instances of other subsystems called the **clients,** which are responsible for interacting with the user.
- ✓ The request for a service is usually done via a remote procedure call mechanism or a common object broker.
- ✓ The client/server architecture is a generalization of the repository architecture.



Advantages

- ✓ Distribution of data is straightforward
- ✓ Makes effective use of networked systems.
- ✓ May require cheaper hardware
- ✓ Easy to add new servers or upgrade existing servers

Disadvantages

- ✓ Redundant management in each server.
- ✓ May require a central registry of names and services: it may be hard to find out what servers and services are available.

5.8.3 Peer-to-Peer

In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

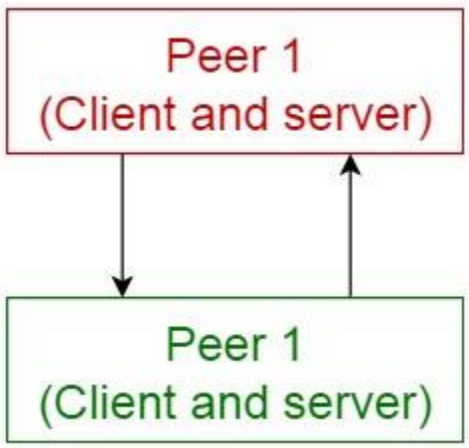


Fig. Peer-to-peer architecture

- ✓ It is a generalization of the client/server architecture in which subsystems can act both as client or as servers, in the sense that each subsystem can request and provide services.
- ✓ The control flow within each subsystem is independent from the others except for synchronizations on requests.
- ✓ Each component acts as its own process and acts as both a client and a server to other peer components.

- ✓ Any component can initiate a request to any other peer component.
- ✓ Characteristics
  - ✦ Scale up well
  - ✦ Increased system capabilities
  - ✦ Peers are distributed in a network, can be heterogeneous, and mutually independent.
  - ✦ Robust in face of independent failures.
  - ✦ Highly scalable

This differs from client/server architectures, in which some computers are dedicated to serving others. Components do not offer the same performance under heavy loads.

Client-Server vs. Peer-to-Peer

Peer-to-peer networks, also called P2P networks, consist of groups of computers (also called nodes or peers) linked together in a network, where peers act as both a client and a server. Peers have equal responsibilities and permissions to work with data. This setup radically differs from the client-server model, which has very defined groups of users and servers. Here are the main differences between the two network models:

- ✓ Client-server networks need a central file server and consequently cost more to implement; peer-to-peer doesn't have that server.
- ✓ Client-server networks delineate between users and providers; peers act as both consumers and providers.
- ✓ Client-server networks offer more levels of security, making them safer. The end-users are responsible for peer-to-peer network security.
- ✓ The more active nodes in a peer-to-peer network, the more its performance suffers. Clientserver networks offer better stability and scalability. The ideal range for P2P networks is two to eight users.
- ✓ Peer-to-peer users can share files faster and easier than on a client-server network.
- ✓ If a client-server network server crashes, everything comes to a halt, but if a single node in a P2P network fails, the rest remains operational.

5.8.4 Event-Based Architecture

This pattern primarily deals with events and has 4 major components; event source, event listener, channel and event bus. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before. Components interact by broadcasting and reacting to events

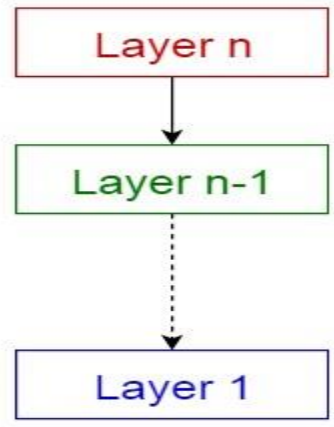
- ✓ Component expresses interest in an event by subscribing to it

When another component announces (publishes) that event has taken place, subscribing components are notified

- ✓ Implicit invocation is a common form of publish-subscribe architecture
- ✓ Registering: subscribing component associates one of its procedures with each event of interest (called the procedure) ❍ Characteristics
- ✓ Strong support for evolution and customization
- ✓ Easy to reuse components in other event-driven systems ✦ Difficult to test

5.8.5 Layered Style

This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.



Layered pattern

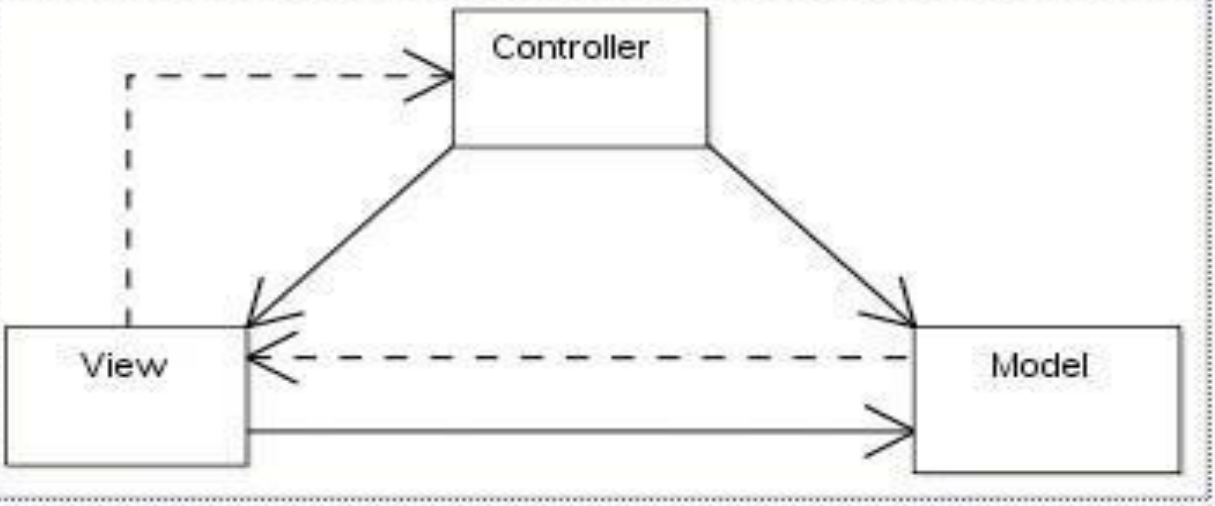
- ❍ Layers are hierarchical
  - ✓ Each layer provides service to the one outside it and acts as a client to the layer inside it.
  - ✓ Layer bridging: allowing a layer to access the services of layers below its lower neighbour.
  - ✓ Each layer has two interfaces.
  - ✓ Upper interface provides services and the lower interface requires services. ❍ The design includes protocols
  - ✓ Explain how each pair of layers will interact
- ❍ **Advantages**
  - ✓ High levels of abstraction
  - ✓ Relatively easy to add and modify a layer
- ❍ Disadvantages
  - ✓ Not always easy to structure system layers

- ✓ System performance may suffer from the extra coordination among layers

5.8.6 Model-View-Controller

In the MVC paradigm, the user input, the modelling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of objects, each specialized for its task.

- ✓ The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application.
- ✓ The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- ✓ The **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The pattern isolates business logic from input and presentation, permitting independent development, testing and maintenance of each.



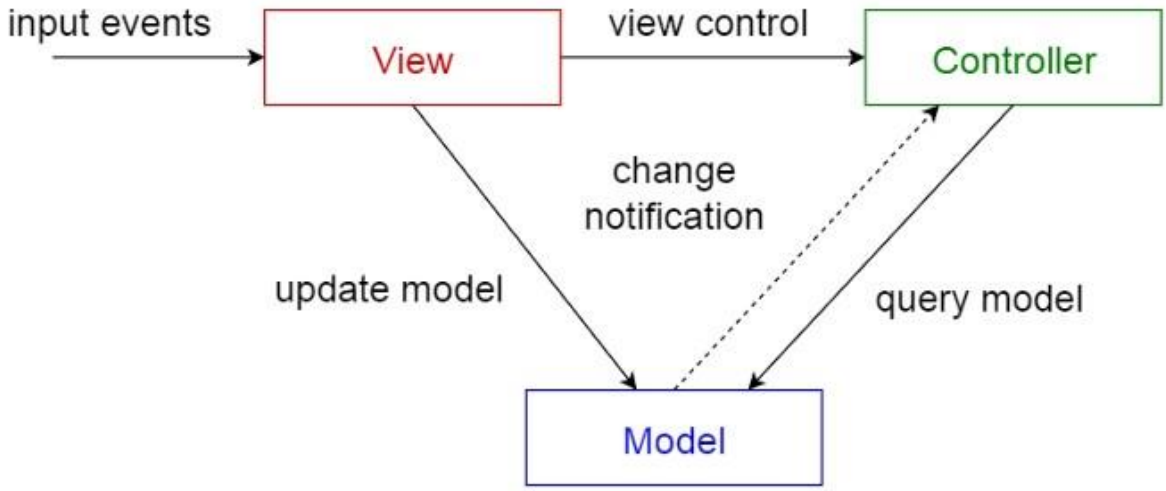
Model-View-Controller concept. Note: The solid line represents a direct association, the dashed an indirect association via an observer (for example).

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts:

1. **Model**: contains the core functionality, data, the business logic, rules, and strategies.
2. **View**: displays the information to the user (more than one view may be defined) or displays the model more than one view and usually has components that allow user to edit or change the model.
3. **Controller**: handles the input from the user.

- ✓ Allows data to flow between the view and the model
- ✓ The controller mediates between the view and model

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.



Model-view-controller pattern

MVC Benefits

- ✓ Clarity of design
  - ✦ easier to implement and maintain
- ✪ Modularity
  - ✦ changes to one don't affect the others
  - ✦ can develop in parallel once you have the interfaces

Chapter 6: Software Testing

Objectives

At the end of this chapter students will be able to

- ☞ Comprehend the concepts of Software Quality, quality assurance and Standards.
- ☞ Study fundamental concepts in software testing
- ☞ Identify the Objective of software Testing
- ☞ Identify the type of software testing
- ☞ Apply effective testing techniques to test a software system
- ☞ Design specific and measurable test cases to ensure software requirements.
- ☞ Demonstrate the integration testing which aims to uncover interaction and compatibility problems as early as possible
- ☞ Integrate testing processes within a continuous delivery model of software development

6.1 Software Testing Terminology

✓ Test Case

A test case, in a practical sense, is a test-related item which contains the following information:

1. **A set of test inputs**: These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
2. **Execution conditions:** These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
3. **Expected outputs:** These are the specified results to be produced by the code under test.

   Example:

- ✓ Name: Search real estate test case
- ✓ Purpose: To test search functionality for a given real estate.

Table 1: Search real estate test case

Input	Expected Results	Actual Results	Pass/Fail
Empty search field	The system should indicate error	As expected	Pass
Wrong value is inserted	The system should indicate error	As expected	Pass
Correct search filed is inserted	The system should display what is found	As expected	Pass

- ✓ **Test Set:**

  A group of related tests is sometimes referred to as a test set.

- ✓ **Test Suite:**

  Test Suite is a high-level concept, grouping together hundreds or thousands of tests related by what they are intended to test

- ✓ **Test Log:**

  Defines the result of test cases in terms of passed/failed after execution of the test case on the application build.

- ✓ **Test Bed**:

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system. This includes the entire testing environment, for example, simulators, emulators, memory checkers, hardware probes, software tools, and all other items needed to support execution of the tests.

- ■ **Test Oracle**:

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed. Examples include a specification (especially one that contains pre- and post-conditions), a design document, and a set of requirements.

A program, or a document that produces or specifies the expected outcome of a test, can serve as an oracle. Examples include a specification (especially one that contains pre- and post-conditions), a design document, and a set of requirements. Other sources are

regression test suites. The suites usually contain components with correct results for previous versions of the software. If some of the functionality in the new version overlaps the old version, the appropriate oracle information can be extracted. A working trusted program can serve as its own oracle in a situation where it is being ported to a new environment. In this case its intended behavior should not change in the new environment.

■ **Error:**

An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of developer, we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.

■ **Faults**

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification. **Faults or defects are sometimes called —bugs.** Use of the latter term trivializes the impact faults have on software quality. Use of the term —defect is also associated with software artifacts such as requirements and design documents. Defects occurring in these artifacts are also caused by errors and are usually detected in the review process.

■ **Failures**

A failure is the inability of a software system or component to perform its required functions within specified performance requirements**.** During execution of a software component or system, a tester, developer, or user observes that it does not produce the expected results.

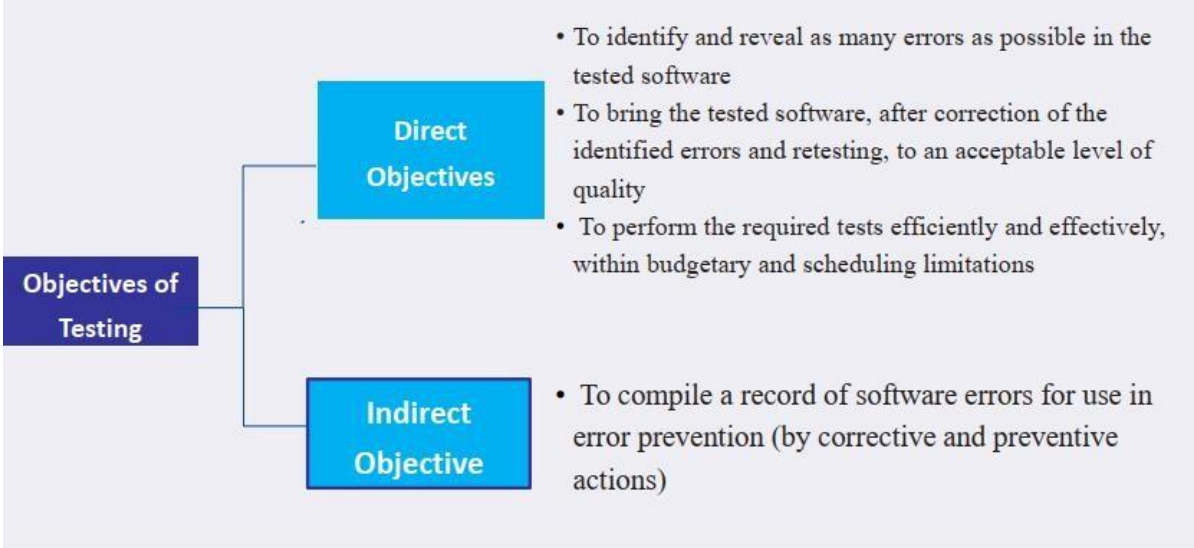
6.2 Observation about Testing

- ✓ "Testing is the process of executing a program with the intention of finding errors". **Myers**
- ✓ "Testing can show the presence of bugs but never their absence" Dijkstra
- ✓ According to **ANSI/IEEE 1059** standard, Testing can be defined as a process of analysing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.

- ✓ Testing ensures that the software behaves and looks exactly like what is mentioned in the requirements specification document, so that when software is delivered to the client there are no arguments about the variation from the original requirements.
- ✓ Testing is the process to execute the program and find the bugs and fix on the spot. It evaluates the quality of the software, and done during the development and after the implementation process.

6.3 Objective of Testing

The main aim of testing is to analyse and examine the performance of a software and to evaluate the errors that occurred during the execution of a software, when the software is running with different inputs in different environment.



When to start testing

An early start to testing reduces the cost, time to rework and error free software that is delivered to the client.

■ However in Software Development Life Cycle (SDLC) testing can be started from the Requirements Gathering phase till the deployment of the software.

■ However it also **depends on the development model that is being used**.

■ For example, in Water fall model, formal testing is conducted in the Testing phase,

■ But in incremental model, testing is performed at the end of every increment/iteration and at the end the whole application is tested.

When to stop testing

Unlike when to start testing, it is difficult to determine when to stop testing, as testing is a **never-ending process** and no one can say that any software is 100% tested. Following are the aspects which should be considered to stop the testing:

- **Completion of Functional and code coverage to a certain point.**
- **Bug rate falls below a certain level and no high priority bugs are identified.**

6.4 Fundamental principles of testing

- ✓ The goal of testing is to find defects before customers find them out.
- ✓ Exhaustive testing is not possible; program testing can only show the presence of defects, never their absence.
- ✓ Testing applies all through the software life cycle and is not an end of-cycle activity.
- ✓ Understand the reason behind the test.
- ✓ Tests develop immunity and have to be revised constantly.
- ✓ Defects occur in convoys or clusters, and testing should focus on these convoys.
- ✓ Testing encompasses defect prevention.
- ✓ Testing is a fine balance of defect prevention and defect detection.
- ✓ Intelligent and well-planned automation is key to realizing the benefits of testing.
- ✓ Testing requires talented, committed people who believe in themselves and work in teams.

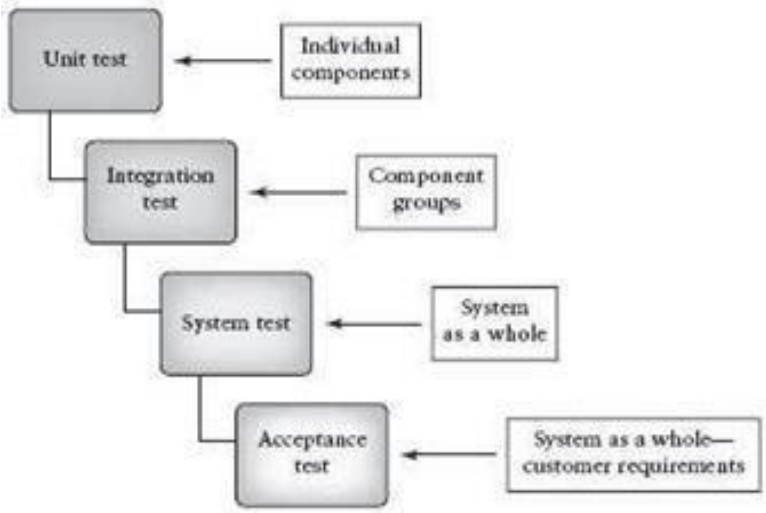
6.5 Levels of Software of Testing

Execution-based software testing, especially for large systems, is usually carried out at different levels. In most cases there will be **3-4 levels**, or major phases of testing: unit test, integration test, system test, and some type of acceptance test as shown in **Figure 6.**

Each of these may consist of one or more sublevels or phases. At each level there are specific testing goals. For example, at unit test a single component is tested.

- ✓ A principal goal is to detect functional and structural defects in the unit.
- ✓ At the integration level several components are tested as a group, and the tester investigates component interactions.

- ✓ At the system level the system as a whole is tested and a principle goal is to evaluate attributes such as usability, reliability, and performance.
- ✓ System test begins when all of the components have been integrated successfully. It usually requires the bulk of testing resources. Laboratory equipment, special software, or special hardware may be necessary, especially for real-time, embedded, or distributed systems. At the system level the tester looks for defects, but the focus is on evaluating performance, usability, reliability, and other quality-related requirements.



The approach used to design and develop a software system has an impact on how testers plan and design suitable tests. There are two major approaches to system development bottom-up, and top- down.

These approaches are supported **by two major types of programming languages:**

- ✓ **Procedure- oriented**
- ✓ **Object-oriented.**

6.5.1 Unit Testing

A unit is the smallest possible testable software component. It can be characterized in several ways.

For example, a unit in a typical procedure-oriented software system:

- ✓ Performs a single cohesive function;
- ✓ can be compiled separately;

- ✓ Is a task in a work breakdown structure (from the manager's point of view);
- ✓ Contains code that can fit on a single page or screen.

A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language. In object-oriented systems both the method and the class/object have been suggested by researchers as the choice for a unit. A unit may also be a small-sized **COTS (commercial off the shelf)** component purchased from an outside vendor that is undergoing evaluation by the purchaser, or a simple module retrieved from an in-house reuse library.

1. Unit Test Planning

A general unit test plan should be prepared. It may be prepared as a component of the master test plan or as a stand-alone plan. It should be developed in conjunction with the master test plan and t h e project plan for each project. Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units. Components of a unit test plan are described in detail the **IEEE Standard for Software Unit Testing.** This standard is rich in information and is an excellent guide for the test planner.

Phase 1: Describe Unit Test Approach and Risks

In this phase of unit testing planning the general approach to unit testing is outlined. The test planner:

a) identifies test risks;

b) describes techniques to be used for designing the test cases for the units;

c) describes techniques to be used for data validation and recording of test results;

d) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

During this phase the planner also identifies completeness requirements what will be covered by the unit test and to what degree (states, functionality, control, and data flow patterns). The planner also identifies termination conditions for the unit tests. Finally, the planner estimates resources needed for unit test, such as hardware, software, and staff, and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit Features to be tested

This phase requires information from the unit specification and detailed design description. The planner determines which features of each unit will be tested, for example: functions, performance requirements, states, and state transitions, control structures, messages, and data flow patterns. If some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assessed. Input/output characteristics associated with each unit should also be identified, such as variables with allowed ranges of values and performance at a certain level.

Phase 3: Add Levels of Detail to the Plan

In this phase the planner refines the plan as produced in the previous two phases. The planner adds new details to the approach, resource, and scheduling portions of the unit test plan. As an example, existing test cases that can be reused for this project can be identified in this phase.

3.1.2. Designing the unit tests

Part of the preparation work for unit test involves unit test design. It is important to specify (i) the test cases (including input data, and expected outputs for each test case), and, (ii) the test procedures (steps required run the tests).

Test case data should be tabularized for ease of use, and reuse. To specifically support objectoriented test design and the organization of test data, Berard has described a test case specification notation. He arranges the components of a test case into a semantic network with parts, Object**_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps.** Each of these items has component parts. In the test design specification Berard also includes lists of relevant states, messages (calls to methods), exceptions, and interrupts. As part of the unit test design process, developers/testers should also describe the relationships between the tests. Test suites can be defined that bind related tests together as a group. All of this test design information is attached to the unit test plan. Test cases, test procedures, and test suites may be reused from past projects if the organization has been careful to store them so that they are easily retrievable and reusable.

2. The Test Harness

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development. The role is of the test harness is shown in **Figure 7** and it is defined as follows:

The auxiliary code developed to support testing of units and components is called a test harness.

The harness consists of drivers that call the target code and stubs that represent modules it calls.

The development of drivers and stubs requires testing resources. The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software. Drivers and stubs can be developed at several levels of functionality.

For example, a driver could have the following options and combinations of options:

call the target unit;

1. do 1, and pass inputs parameters from a table;
2. do 1, 2, and display parameters;
3. do 1, 2, 3 and display results (output parameters).

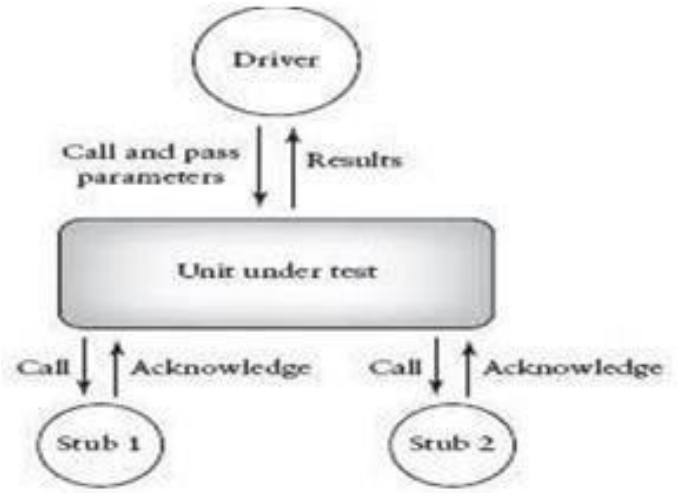


Figure 8:Test Harness

The stubs could also exhibit different levels of functionality. For example, a stub could:

1. display a message that it has been called by the target unit;
2. do 1, and display any input parameters passed from the target unit;
3. do 1, 2, and pass back a result from a table;
4. do 1, 2, 3, and display result from table.

3. Running the unit tests and recording results

Unit tests can begin when

- The units become available from the developers (an estimation of availability is part of the test plan),
- The test cases have been designed and reviewed, and
- The test harness, and any other supplemental supporting tools, are available.

The testers then proceed to run the tests and record results. The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in **Figure 9 .** These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress. It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test. If the test is failed, the nature of the problem should be recorded in what is sometimes called a test incident report.

Other likely causes that need to be carefully investigated by the tester are the following:

- a fault in the test case specification (the input or the output was not specified correctly);
- a fault in test procedure execution (the test should be rerun);
- a fault in the test environment (perhaps a database was not set up properly);
- a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

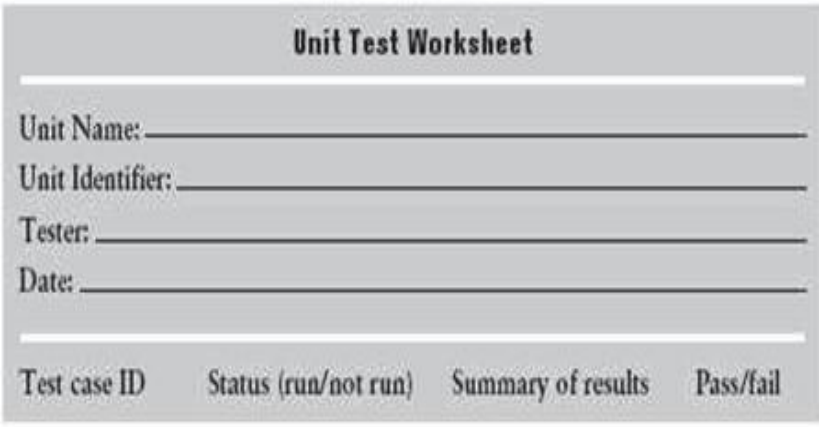


Figure 9: Unit Test worksheet

The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by the unit test plan.

6.5.2 Integration Testing

Integration test for procedural code has **two major goals:**

1. To detect defects that occur on the interfaces of units;
2. To assemble the individual units into working subsystems and finally a complete system that is ready for system test.

With a few minor exceptions, integration test should only be performed on units that have been reviewed and have successfully passed unit testing. A tester might believe erroneously that since a unit has already been tested during a unit test with a driver and stubs, it does not need to be retested in combination with other units during integration test. However, a unit tested in isolation may not have been tested adequately for the situation where it is combined with other modules.

Integration testing works best as an iterative process in procedural oriented system. One unit at a time is integrated into a set of previously integrated modules which have passed a

set of integration tests. The interfaces and functionally of the new unit in combination with the previously integrated units is tested. When a subsystem is built from units integrated in this stepwise manner, then performance, security, and stress tests can be performed on this subsystem.

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios. These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests. The strategy for integration should be defined. For procedural-oriented system the order of integration of the units of the units should be defined. This depends on the strategy selected. Consider the fact that the testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly with the integration test cases. For object-oriented systems a working definition of a cluster or similar construct must be described, and relevant test cases must be specified. In addition, testing resources and schedules for integration should be included in the test plan.

6.5.3 System Testing

When integration tests are completed, a software system has been assembled and its major subsystems have been tested. At this point the developers/ testers begin to test it as a whole. System test planning should begin at the requirements phase with the development of a master test plan and requirements-based (black box) tests. System test planning is a complicated task. There are many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases, and test procedures.

System testing itself requires a large number of resources. The goal is to ensure that the system performs according to its requirements. System test evaluates both functional behaviour and quality requirements such as reliability, usability, performance and security. This phase of testing is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, and deadlocks, problems with interrupts and exception handling, and ineffective memory usage. After system test the software will be turned over to users for evaluation during acceptance test or alpha/beta test. The organization will want to be sure that the quality of the software has been

measured and evaluated before users/clients are invited to use the system. In fact, system test serves as a good rehearsal scenario for acceptance test.

Because system test often requires many resources, special laboratory equipment, and long test times, it is usually performed by a team of testers. The best scenario is for the team to be part of an independent testing group. The team must do their best to find any weak areas in the software; therefore, it is best that no developers are directly involved.

There are several types of system tests as follows:

- ✓ Functional testing
- ✓ Performance testing
- ✓ Stress testing
- ✓ Configuration testing
- ✓ Security testing
- ✓ Recovery testing
- ✓ Web Application Testing

8. Functional Testing

System functional tests have a great deal of overlap with acceptance tests. Very often the same test sets can apply to both. Both are demonstrations of the system's functionality. Functional tests at the system level are used to ensure that the behaviour of the system adheres to the requirements specification. All functional requirements for the system must be achievable by the system. For example, if a personal finance system is required to allow users to set up accounts, add, modify, and delete entries in the accounts, and print reports, the function-based system and acceptance tests must ensure that the system can perform these tasks. Clients and users will expect this at acceptance test time. Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function. Improper and illegal inputs must also be handled by the system. System behaviour under the latter circumstances tests must be observed. All functions must be tested. In fact, the tests should focus on the following goals.

- ✓ All types or classes of legal inputs must be accepted by the software.
- ✓ All classes of illegal inputs must be rejected (however, the system should remain available).
- ✓ All possible classes of system output must exercise and examined.

- ✓ All effective system states and state transitions must be exercised and examined.
- ✓ All functions must be exercised.

9. Performance Testing

An examination of a requirements document shows that there are two major types of requirements:

Functional requirements. Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests. **Quality requirements.** There are non-functional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays. The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test whether there are any hardware or software factors that impact on the system's performance. Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources. For example, testers may find that they need to reallocate memory pools, or to modify the priority level of certain system operations. Testers may also be able to project the system's future performance levels. This is useful for planning subsequent releases.

10. Stress Testing

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing. For example, if an operating system is required to handle 10 interrupts/second and the load causes 20 interrupts/second, the system is being stressed. The goal of stress test is to try to break the system; find the circumstances under which it will crash. This is sometimes called "**breaking the system**". An everyday analogy can be found in the case where a suitcase being tested for strength and endurance is stomped on by a multiton elephant!

Stress testing is important from the user/client point of view. When systems operate correctly under conditions of stress then clients have confidence that the software can perform as required. Beizer suggests that devices used for monitoring stress situations provide users/clients with visible and tangible evidence that the system is being stressed.

11. Configuration Testing

Typical software systems interact with hardware devices such as disc drives, tape drives, and printers. Many software systems also interact with multiple CPUs, some of which are redundant. Software that controls real-time processes, or embedded software also interfaces with devices, but these are very specialized hardware items such as missile launchers, and nuclear power device sensors.

In many cases, users require that devices be interchangeable, removable, or reconfigurable. For example, a printer of type X should be substitutable for a printer of type Y, CPU A should be removable from a system composed of several other CPUs, sensor A should be replaceable with sensor B. Very often the software will have a set of commands, or menus, that allows users to make these configuration changes. Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur.

According to Beizer configuration testing has the following objectives:

- ✓ Show that all the configuration changing commands and menus work properly.
- ✓ Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.
- ✓ Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

12. Security Testing

Designing and testing software systems to ensure that they are safe and secure is a big issue facing software developers and test specialists. Recently, safety and security issues have taken on additional importance due to the proliferation of commercial applications for use on the Internet. If Internet users believe that their personal information is not secure and is available to those with intent to do harm, the future of e-commerce is in peril! Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services. Users/clients should be encouraged to make sure their security needs are clearly known at requirements time, so that security issues can be addressed by designers and testers.

Computer software and data can be compromised by:

a) Criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy;

b) Errors on the part of honest developers/maintainers who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge.

Both criminal behavior and errors that do damage can be perpetuated by those inside and outside of an organization. Attacks can be random or systematic. Damage can be done through various means such as:

- ✓ viruses;
- ✓ Trojan horses;
- ✓ trap doors;
- ✓ illicit channels.

The effects of security breaches could be extensive and can cause:

- ✓ loss of information;
- ✓ corruption of information;
- ✓ misinformation;
- ✓ privacy violations;
- ✓ denial of service.

A password checker can enforce any rules the designers deem necessary to meet security requirements.

Password checking and examples of other areas to focus on during security testing are described below.

- ✓ **Password Checking:** Test the password checker to ensure that users will select a password that meets the conditions described in the password checker specification. Equivalence class partitioning and boundary value analysis based on the rules and conditions that specify a valid password can be used to design the tests.
- ✓ **Legal and Illegal Entry with Passwords:** Test for legal and illegal system/data access via legal and illegal passwords.

- ✓ **Password Expiration:** If it is decided that passwords will expire after a certain time period, tests should be designed to insure the expiration period is properly supported and that users can enter a new and appropriate password.
- ✓ **Encryption:** Design test cases to evaluate the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
- ✓ **Browsing:** Evaluate browsing privileges to ensure that unauthorized browsing does not occur. Testers should attempt to browse illegally and observe system responses. They should determine what types of private information can be inferred by both legal and illegal browsing.
- ✓ **Trap Doors:** Identify any unprotected entries into the system that may allow access through unexpected channels (trap doors). Design tests that attempt to gain illegal entry and observe results. Testers will need the support of designers and developers for this task. In many cases an external "tiger team" as described below is hired to attempt such a break into the system.
- ✓ **Viruses:** Design tests to ensure that system virus checkers prevent or curtail entry of viruses into the system. Testers may attempt to infect the system with various viruses and observe the system response. If a virus does penetrate the system, testers will want to determine what has been damaged and to what extent.

13. Recovery Testing

Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses. This type of testing is especially important for transaction systems, for example, on-line banking software.

Beizer advises that testers focus on the following areas during recovery testing:

1. **Restart.** The current system state and transaction states are discarded. The most recent checkpoint record is retrieved and the system initialized to the states in the checkpoint record. Testers must insure that all transactions have been reconstructed correctly and that all devices are in the proper state. The system should then be able to begin to process new transactions.
2. **Switchover.** The ability of the system to switch to a new processor must be tested. Switchover is the result of a command or a detection of a faulty processor by a

monitor. In each of these testing situations all transactions and processes must be carefully examined to detect: **a)** loss of transactions;

b) merging of transactions;

c) incorrect transactions;

d) An unnecessary duplication of a transaction.

A good way to expose such problems is to perform recovery testing under a stressful load. Transaction inaccuracies and system crashes are likely to occur with the result that defects and design flaws will be revealed.

14. Regression Testing

Regression testing is not a level of testing, but it is the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes. Regression testing can occur at any level of test, for example, when unit tests are run the unit may pass a number of these tests until one of the tests does reveal a defect. The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality.

Regression tests are especially important when multiple software releases are developed. Users want new capabilities in the latest releases, but still expect the older capabilities to remain in place. This is where regression testing plays a role. Test cases, test procedures, and other test- related items from previous releases should be available so that these tests can be run with the new versions of the software. Automated testing tools support testers with this very time- consuming task.

❖ Web Application Testing

Web testing is the name given to software testing that focuses on web applications [2]. Web application testing is a mandatory skill for software test engineers these days. It checks bottlenecks and performance leakage in the software or web application to be tested.

Web application testing is a quality assurance activity evaluating the system under test (SUT) by observing its execution to reveal failures [3]. When the SUT's external behavior differs from what we expect of the SUT according to its requirements or some other description of the expected behavior, a failure is discovered [4].

15. Type of Web Application Testing

Web application testing can be divided into two main categories, manual testing, and automated web application testing [5]. Both categories have their strengths and weaknesses.

A. Manual Testing

Manual testing is a testing process that is carried out manually to find defects without the usage of tools or automation scripting. A test plan document is created by the test lead which describes the detailed and systematic approach to testing a software application and which the tester has to be followed. Typically, the test plan includes a complete understanding of what the ultimate workflow will be. To ensure the completeness of testing, test cases or test scenarios are created [5].

The tester runs the test cases manually to see whether they have any defects. In manual testing, the tester has to pretend as an application user and use all kinds of functions available to find out its right behavior [6]. Manual testing can't be used again, no facility for scripting, and few errors may remain undetected. In addition, manually performed software testing is a time-consuming and burdensome task. Therefore, authors have developed various automated methods and tools to help developers test the quality of a web application.

B. Automated Testing

Automated testing focuses on replacing manual human activities with systems or devices that enhance the efficiency of the software testing process. It uses tools, scripts, and software to perform test cases by repeating pre-defined actions. It is suitable for regression testing, performance testing, load testing, or highly repeatable functional test cases.

The test case is a crucial component of the testing process. A test case outlines the conditions under which SUT detects a failure. When a test case reveals a failure, it is successful. A test case embodies the input values needed to execute the SUT [4]. Therefore, test case inputs vary in nature, ranging from user inputs to method calls with the test-case values as parameters. To evaluate the results of test cases, testers must first know where an element is situated, then conduct an action on the web element and examine the result.

6.5.4 Acceptance Testing

Acceptance testing is the last phase of testing and is used to assess whether or not the final piece of software is ready for delivery. It involves ensuring that the product is incompliance

with all of the original business criteria and that it meets the end user's needs. This requires the product be tested both internally and externally, meaning you'll need to get it into the hands of your end users for beta testing along with those of your QA team. Beta testing is key to getting real feedback from potential customers and can address any final usability concerns. When software is being developed for a specific client, acceptance tests are carried out after system testing. The acceptance tests must be planned carefully with input from the client/users.

Acceptance test cases are based on requirements. The user manual is an additional source for test cases. System test cases may be reused. The software must run under real-world conditions on operational hardware and software. The software-under-test should be stressed. For continuous systems the software should be run at least through a 25-hour test cycle. Conditions should be typical for a working day. Typical inputs and illegal inputs should be used and all major functions should be exercised. If the entire suite of tests cannot be run for any reason, then the full set of tests needs to be rerun from the start.

Acceptance tests are a very important milestone for the developers. At this time the clients will determine if the software meets their requirements. Contractual obligations can be satisfied if the client is satisfied with the software. Development organizations will often receive their final payment when acceptance tests have been passed.

Acceptance testing can be divided in to two. i.e. **Beta testing** and **Alpha testing**. Beta User testing of a completed information system using real data in the real user environment. Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality. Beta testing reduces product failure risks and provides increased quality of the product through customer validation. Beta testing is performed by Clients or End.

Users who are not employees of the organization. Alpha testing is a type of user acceptance testing or internal acceptance testing conducted by the testing team at the developer's site. This is done to ensure that the application's business requirements are met before releasing it to the client or the end-user. It is carried out in the later phase of software development using both black-box and white-box testing techniques.

After acceptance testing the client will point out to the developers which requirement have/have not been satisfied. Some requirements may be deleted, modified, or added due to changing needs. If the client has been involved in prototype evaluations, then the changes

may be less extensive. If the client is satisfied that the software is usable and reliable, and they give their approval, then the next step is to install the system at the client's site. If the client's site conditions are different from that of the developers, the developers must set up the system so that it can interface with client software and hardware. Retesting may have to be done to ensure that the software works as required in the client's environment**.** Table 1 shows the conclusion of level of software testing.

Table 5: Level of Software Testing Conclusion

	Unit Testing	Integration	System	Acceptance
Why	To ensure the code is developed correctly	To ensure the ties between the system component functions as required	To ensure the whole system worked when integrated	To ensure customers and end user expectation met
Who	Developer/Technical Architects	Developer/Technical Architects	SDET/Manual QA, BA/ Product owner	Developer, SDET/Manual QA,/Product owner, end user
What	All new code	New web service components, controllers	Scenario testing User flow	Verifying Acceptance testing
When	As soon as a new code is written	As soon as a new component is written	When the product is complete	When the product is shipped
Where	Local Dev+ Continuous integration	Local Dev+ CI (Part of the build)	Staging Environment	CI/ Test Environment
How	Automated, Junit, TestNG	Automated, Soup UI	Automated (web driver)	Automated(Cucumber)

6.6 Classification of software Testing Techniques

Testing strategies define the approaches for designing test cases. They can be responsibility-based (also known as a black box), an implementation-based, or hybrid (also known as a grey box).

Classification based on the criterion to measure the adequacy of a set of test cases:

- ✓ **Coverage-based testing**: How many statements or requirements have been tested
- ✓ **Fault-based testing**: how many seeded faults are found
- ✓ **Error-based testing**: focus on error-prone points

6.6.1 White Box Strategies

White box strategies design test cases based on a coded representation of the component under test and of a coverage model that specifies the parts of the representation that must be exercised by a test suite. As an example, in the case of traditional software the control flow

graph is a typical test model, while statement coverage, branch coverage, or basis-path coverage are possible code coverage models [14].

6.6.2 Black Box Strategies

Black box techniques do not require knowledge of software implementation items under test since test cases are designed based on an item's specified or expected functionality. One main issue with black-box testing of Web applications is the choice of a suitable model for specifying the behavior of the application to be tested and to derive test cases. Indeed, this behavior may significantly depend on the state of data managed by the application and on user input, with the consequence of a state explosion problem even in the presence of applications implementing a few simple requirements.

6.6.3 Grey Box Strategies

Grey box testing strategies combine black box and white box testing approaches to design test cases: they aim at testing a piece of software against its specification but using some knowledge of its internal workings.

Review Questions

1. Which of the following should **not** normally be the direct objective for software testing?
   A. To find faults in the software
   B. To supply records of software errors to be used for error prevention
   C. To assess whether the software is ready to release
   D. To demonstrate that the software doesn't work
2. When does integration testing conducted?
   A. As soon as a new code is written
   B. When a new component is written
   C. When the product is complete
   D. When the product is shipped
3. When should testing be stopped?
   A. Bug rate falls below a certain level and no high priority bugs are identified
   B. Deadline (Testing Release)
   C. Completion of Functional and code coverage to a certain point
   D. All
4. Which one of the following statements correctly describes the key principles of software testing?
   A. By using automated testing, it is possible to test everything.

B. With sufficient effort and tool support, exhaustive testing is feasible for all software.

C. It is impossible to test all input and precondition combinations in a system D. The purpose of testing is to prove the absence of a defect

Bibliography

[1] Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, Object-Oriented Software Engineering, Wokingham: Addison-Wesley., 1993.

[2] Ja m e s Ru m b a u g h,Ivar Ja cobs on and Gra dy B o o c h, The Unified Modeling Language Reference Manual, 1998.