

Software Engineering



Hosea D. Shimwela
Computer Studies Department
Dar es Salaam Institute of Technology (DIT)

October, 2024

Outline:



- 1) **Software Definitions, Characteristics and Applications**
- 2) **Software Development Life Cycle (SDLC)**
- 3) **Software Process Model**
- 4) **Software Requirement Characteristics**
- 5) **Design Concepts and Principles**
- 6) **Software Testing Techniques**
- 7) **References**

1) Software Definitions, characteristics and applications

What is a Software?

- Is a computer programs and associated **documentations**

Software products may be

- **Generic** - developed to be sold to a range of different customers
- **Custom** - developed for a single customer according to their specification

What is software engineering?

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

1) Software Definitions, characteristics and applications

What is software engineering?

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

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

So, we can define **software engineering** as a detailed study of engineering to the design, development and maintenance of software. Software engineering was introduced to address the issues of **low-quality software projects**. Problems arise when a software generally exceeds **timelines, budgets, and reduced levels of quality**.

1) Software Definitions, characteristics and applications

Software characteristics

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

1) Software Definitions, characteristics and applications

Software characteristics

The following are software characteristics:

- 1) Operational:** This tells how good a software works on operations like budget, usability, efficiency, correctness, functionality, dependability, security and safety.
- 2) Transitional:** Transitional is important when an application is shifted from one platform to another. So, **portability**, **reusability** and **adaptability** come in this area.
- 3) Maintenance:** This specifies how good a software works in the changing environment. **Modularity**, **maintainability**, **flexibility** and **scalability** come in maintenance part.
- 4) Software is developed or engineered:** Software is developed or engineered and is not manufactured in the classical sense. Some similarities exist between software development and hardware manufacture, however, the two activities are fundamentally different.

1) Software Definitions, characteristics and applications

Software characteristics

5) Software doesn't "wear out": The hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (Ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many others.

6) Most software is custom-built: Rather than being assembled from existing components, most software are custom built and have specific characteristics.

1) Software Definitions, characteristics and applications

Software applications

- Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (exceptions for [neural network software \(AI\)](#)).
- [Information contents](#) are the important factors in determining the nature of a software application. Content refers to incoming and outgoing information. For example, many business applications use structured input data (a [database](#)) and produce formatted "[reports](#)."
- However, nowadays we have more [unstructured big data](#) from [social media](#), live streaming, sensors (radar, camera, etc)

1) Software Definitions, characteristics and applications

Software applications

The following are some of the types of software applications:

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

1) Software Definitions, characteristics and applications

Software applications

2) Real time software: These software are used to monitor, control and analyze real world events as they occur. An example may be software required for [weather forecasting](#). Such software will [gather and process the status of temperature](#), humidity and other environmental parameters to for-cast the weather.

3) Engineering and scientific software: These software applications range from [astronomy](#) to [volcanology](#), from [automotive analysis](#) to [space shuttle orbital dynamics](#), and from molecular biology to [automated manufacturing](#). However, modern applications within the engineering/scientific area are moving towards system simulation, and other interactive applications.

1) Software Definitions, characteristics and applications

Software applications

4) Embedded software: This type of software is placed in “Read-Only- Memory (ROM)” of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signaling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as **intelligent software**.

5) Business software: This is the largest application area. The software designed to process business applications is called business software. Business software could be **payroll**, **file monitoring system**, **employee management**, and **account management**. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software.

1) Software Definitions, characteristics and applications

Software applications

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

7) Personal computer software: The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organizations are concentrating their effort here due to large customer base.

8) Artificial intelligence software: Artificial Intelligence software makes use of non-numerical algorithms to solve complex problems that are not responsive to computation or straight forward analysis. Examples are artificial neural network, signal processing software etc.

1) Software Definitions, characteristics and applications

Software applications

Legacy Software:

- Legacy software is a software that has been around for a long time and still fulfills a business need. It is mission critical tied to a particular version of an **operating system** or **hardware model (vendor lock-in)** that has gone end-of-life. **Generally the lifespan of the hardware is shorter than that of the software.**
- Today, a growing population of **legacy programs** is forcing many companies to pursue software reengineering strategies. The term legacy programs stand for older, often poorly designed and documented software that is business critical and must be supported over many years. Some legacy systems have relatively **solid program architecture**, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

2) Software Development Life Cycle (SDLC)

What is the SDLC?

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

2) Software Development Life Cycle (SDLC)

Benefits of the SDLC

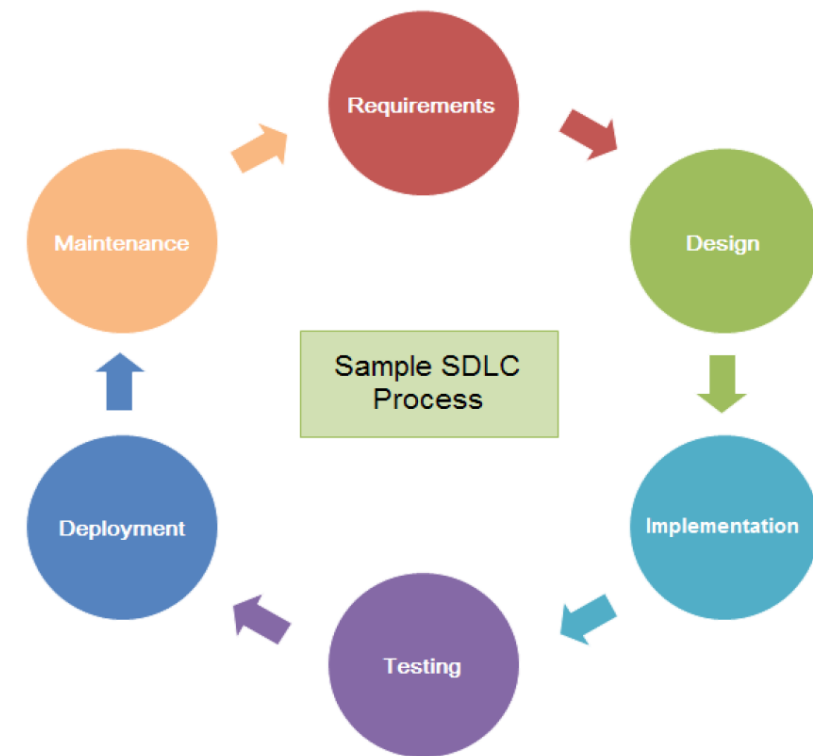
- SDLC in your software development process offers numerous benefits, including:
 - **Higher Quality Software:** The structured approach of the SDLC helps ensure that the software meets user requirements and is reliable, efficient, and secure.
 - **Better Collaboration:** The SDLC provides a common framework for developers, designers, and other stakeholders to work together effectively.
 - **Improved Project Management:** By breaking down the development process into manageable phases, the SDLC makes it easier to plan, track, and control the progress of a project.
 - **Reduced Risks:** The SDLC helps identify and address potential issues and risks early in the development process, minimizing their impact on the project.

2) Software Development Life Cycle (SDLC)

SDLC phases

- A typical [Software Development Life Cycle \(SDLC\)](#) consists of the following phases:

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



2) Software Development Life Cycle (SDLC)

1) Requirement gathering:

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

2) Software Development Life Cycle (SDLC)

2) System Analysis:

- In analysis Phase, once the requirement gathering and analysis is done, the next step is to define and document the product requirements and get them approved by the customer (Client).
- This is done through SRS (Software Requirement Specification) document.
- SRS consists of all the product requirements to be designed and developed during the project life cycle.
- Key people involved in this phase are [Project Manager](#), [Business Analyst](#) and [Senior members](#) of the Team. The outcome of this phase is Software Requirement Specification.

2) Software Development Life Cycle (SDLC)

3) Design:

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

2) Software Development Life Cycle (SDLC)

4) Development/Implementation or Coding:

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

2) Software Development Life Cycle (SDLC)

5) Testing:

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

2) Software Development Life Cycle (SDLC)

6) Deployment:

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

2) Software Development Life Cycle (SDLC)

7) Maintenance:

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

3) Software Process Model

Software Process:

- Software Processes is a coherent set of activities for specifying, designing, implementing and testing software systems. A software process model is an abstract representation of a process that presents a description of a process from some particular perspective.
- There are many different software processes but all involve:
 - a) **Specification** – defining what the system should do
 - b) **Design and implementation** – defining the organization of the system and implementing the system
 - c) **Validation** – checking that it does what the customer wants
 - d) **Evolution** – changing the system in response to changing customer needs

3) Software Process Model

Types of Software Process Model:

- When we describe and discuss processes, we usually think about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities. There are different models available, for example:
 - Waterfall model
 - V model
 - Incremental model
 - Prototyping model
 - RAD model
 - Agile model
 - etc.

Assignment 1



Group work

- A. Describe in details, if possible, with diagram, the given Software Process Model in your group?
- B. Mention advantages and disadvantages of using the given software process model
- C. When is appropriate to use the given Software Process Model?
- D. Your descriptions should not exceed 5 pages
- E. Submission/presentation is on **week 10**

4) Software Requirement Characteristics:

Software Requirements:

- Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct and well-defined.
- A complete Software Requirement Specifications must be:
 - 1) Clear
 - 2) Correct
 - 3) Consistent
 - 4) Coherent
 - 5) Comprehensible
 - 6) Modifiable
 - 7) Verifiable
 - 8) Prioritized
 - 9) Unambiguous
 - 10) Traceable
 - 11) Credible source

4) Software Requirement Characteristics:

Software Requirements:

We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.

Broadly software requirements should be categorized in two categories: functional and Non-functional requirements.

- **Functional Requirements**

Requirements, which are related to functional aspect of software fall into this category. They define functions and functionality within and from the software system. Function examples are:

- 1) Search option given to user to search from various invoices
- 2) User should be able to mail any report to management
- 3) Users can be divided into groups and groups can be given separate rights
- 4) Should comply business rules and administrative functions
- 5) Software is developed keeping downward compatibility intact

4) Software Requirement Characteristics:

Software Requirements:

- **Non-Functional Requirements**

Requirements, which are not related to functional aspect of software, fall into this category.

They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements examples include:

- 1) Security
- 2) Logging
- 3) Storage
- 4) Configuration
- 5) Performance
- 6) Cost
- 7) Interoperability
- 8) Flexibility
- 9) Disaster recovery
- 10) Accessibility

4) Software Requirement Characteristics:

Software Requirements:

Requirements are categorized logically as:

- **Must Have:** Software cannot be said operational without them.
- **Should have:** Enhancing the functionality of software.
- **Could have:** Software can still properly function without these requirements.
- **Wish list:** These requirements do not map to any objectives of software.

While developing software, '**Must have**' must be implemented, '**Should have**' is a matter of debate with stakeholders and negotiation, whereas '**could have**' and '**wish list**' can be kept for software updates.

4) Software Requirement Characteristics:

User Interface (UI) requirements:

UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is:

- a) easy to operate
- b) quick in response
- c) effectively handling operational errors
- d) providing simple yet consistent user interface

User acceptance majorly depends upon how user can use the software. UI is the only way for users to perceive the system. A well performing software system must also be equipped with attractive, clear, consistent and responsive user interface. Otherwise the functionalities of software system can not be used in a convenient way. A system is said to be good if it provides means to use it efficiently.

4) Software Requirement Characteristics:

User Interface (UI) requirements:

User interface requirements are briefly mentioned below:

- Content presentation
- Easy Navigation
- Simple interface
- Responsive
- Consistent UI elements
- Feedback mechanism
- Default settings
- Purposeful layout
- Strategical use of color and texture.
- Provide help information
- User centric approach
- Group based view settings

4) Software Requirement Characteristics:

Software Metrics and Measures:

- Software Measures can be understood as a process of quantifying and symbolizing various attributes and aspects of software.
- Software Metrics provide measures for various aspects of software process and software product.
- Software measures are fundamental requirement of software engineering. They not only help to control the software development process but also aid to keep quality of ultimate product excellent.
- According to Tom DeMarco, a (Software Engineer), “*You cannot control what you cannot measure.*” By his saying, it is very clear how important software measures are.

4) Software Requirement Characteristics:

Software Metrics and Measures:

Let us see some software metrics:

- **Size Metrics:** LOC (Lines of Code), mostly calculated in thousands of delivered source code lines, denoted as LOC.
- **Function Point Count:** is a measure of the functionality provided by the software. Function Point count defines the size of functional aspect of software.
- **Complexity Metrics:** McCabe's Cyclomatic complexity quantifies the upper bound of the number of independent paths in a program, which is perceived as complexity of the program or its modules. It is represented in terms of graph theory concepts by using control flow graph.

4) Software Requirement Characteristics:

Software Metrics and Measures:

- **Quality Metrics:** Defects, their types and causes, consequence, intensity of severity and their implications define the quality of product. The number of defects found in development process and number of defects reported by the client after the product is installed or delivered at client-end, define quality of product.
- **Process Metrics:** In various phases of SDLC, the methods and tools used, the company standards and the performance of development are software process metrics.
- **Resource Metrics:** Effort, time and various resources used, represents metrics for resource measurement.

4) Software Requirement Characteristics:

Developing Use-Cases:

Before we start working on any project, it is very important that we are very clear on what we want to do and how do we want to do it.

- What are Use Cases?

In software and systems engineering, a use case is a list of actions or event steps, typically defining the interactions between a role (known in the Unified Modeling Language (UML) as an **actor**) and a system, to achieve a goal.

- The **actor** can be a human, an external system, or a time. In systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals.
- Another way to look at it is a use case describes a way in which a real-world actor interacts with the system. In a system use case you include high-level implementation decisions.

4) Software Requirement Characteristics:

Developing Use-Cases:

- What is the importance of Use Cases?

Use cases have been used extensively over the past few decades. The advantages of Use cases includes:

- 1) The list of goal names provides the shortest summary of what the system will offer
- 2) It gives an overview of the roles of each and every component in the system. It will help us in defining the role of users, administrators etc.
- 3) It helps us in extensively defining the user's need and exploring it as to how it will work.
- 4) It provides solutions and answers to many questions that might pop up if we start a project unplanned.

4) Software Requirement Characteristics:

Developing Use-Cases:

- How to plan use case?

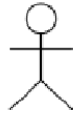


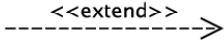
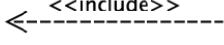
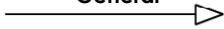
Following example will illustrate on how to plan use cases:

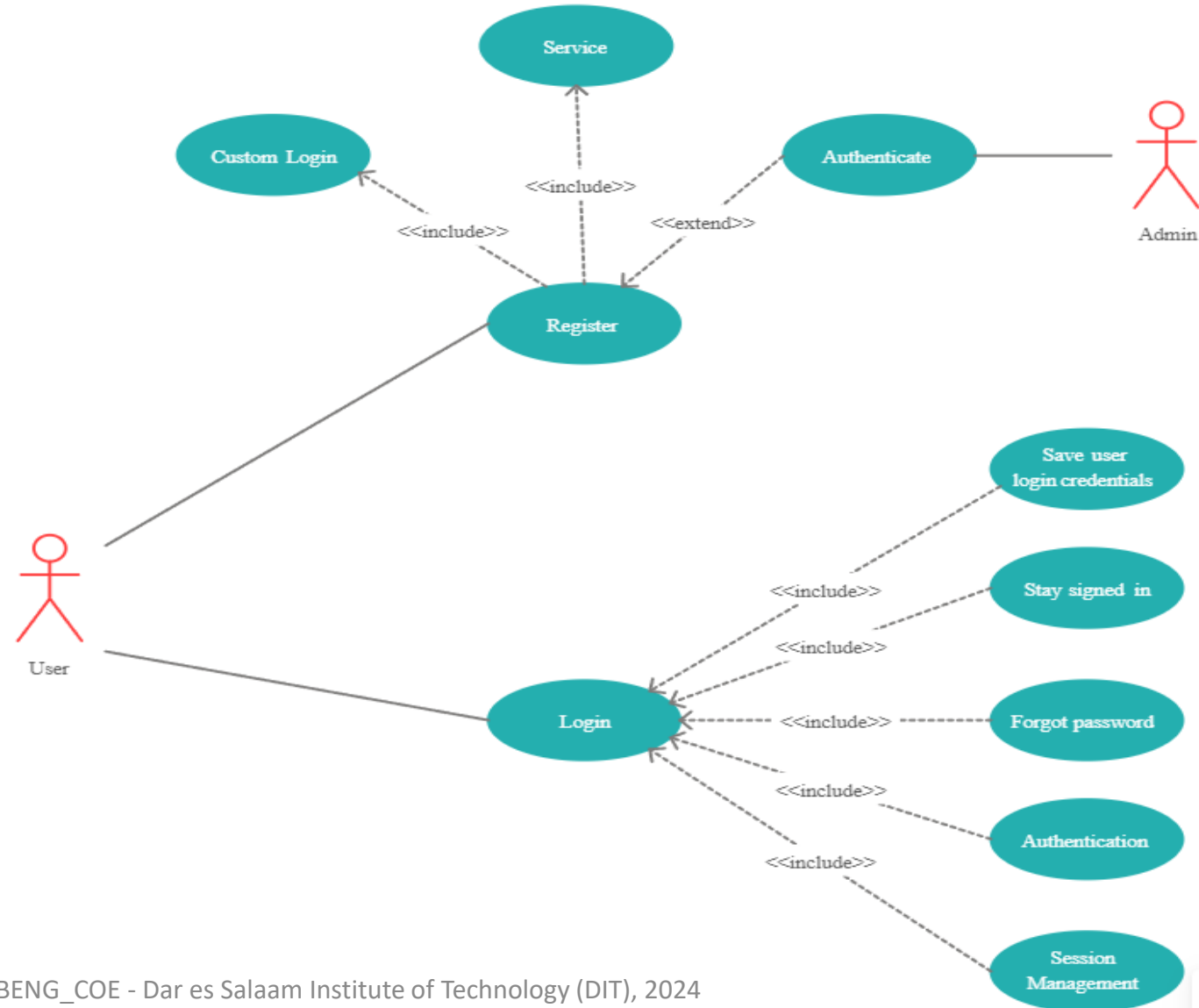
- **Use Case:** What is the main objective of this use case. For e.g. Adding a software component, adding certain functionality, etc.
- **Primary Actor:** Who will have the access to this use case. In the above examples, administrators will have the access.
- **Scope:** Scope of the use case
- **Level:** At what level the implementation of the use case will be.
- **Flow:** What will be the flow of the functionality that needs to be there. More precisely, the work flow of the use case.

4) Software Requirement Characteristics:

Use-Case Diagram:

Key:

Symbol	Reference Name
	Actor
	Use case
Association   <<extend>>  <<include>> General 	Relationship



5) Design Concepts and Principles:

Design Principles:

- Some of the commonly followed design principles are as following:
 - a) **Software design should correspond to the analysis model:** We must know how the design model satisfies all the requirements represented by the analysis model.
 - b) **Choose the right programming paradigm:** 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.
 - c) **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.
 - d) **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) Design Concepts and Principles:

Design Principles:

- e) **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.
- f) **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.
- g) **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.
- h) **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.

5) Design Concepts and Principles:

Design Principles:

i) **Designing for testability:** A common practice is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed and handed over to the testers who subsequently determine whether the software is fit for distribution to customer. However, it has become apparent that the process of separating testing is seriously faulty. 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.

j) **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 an 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.

5) Design Concepts and Principles:

Software Design Concepts:

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

- **Abstraction:** Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components.
- **Abstraction is defined 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.

5) Design Concepts and Principles:

Software Design Concepts:

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

1. **Functional abstraction:** This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.
2. **Data abstraction:** This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.

5) Design Concepts and Principles:

Software Design Concepts:

3. Control abstraction: This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

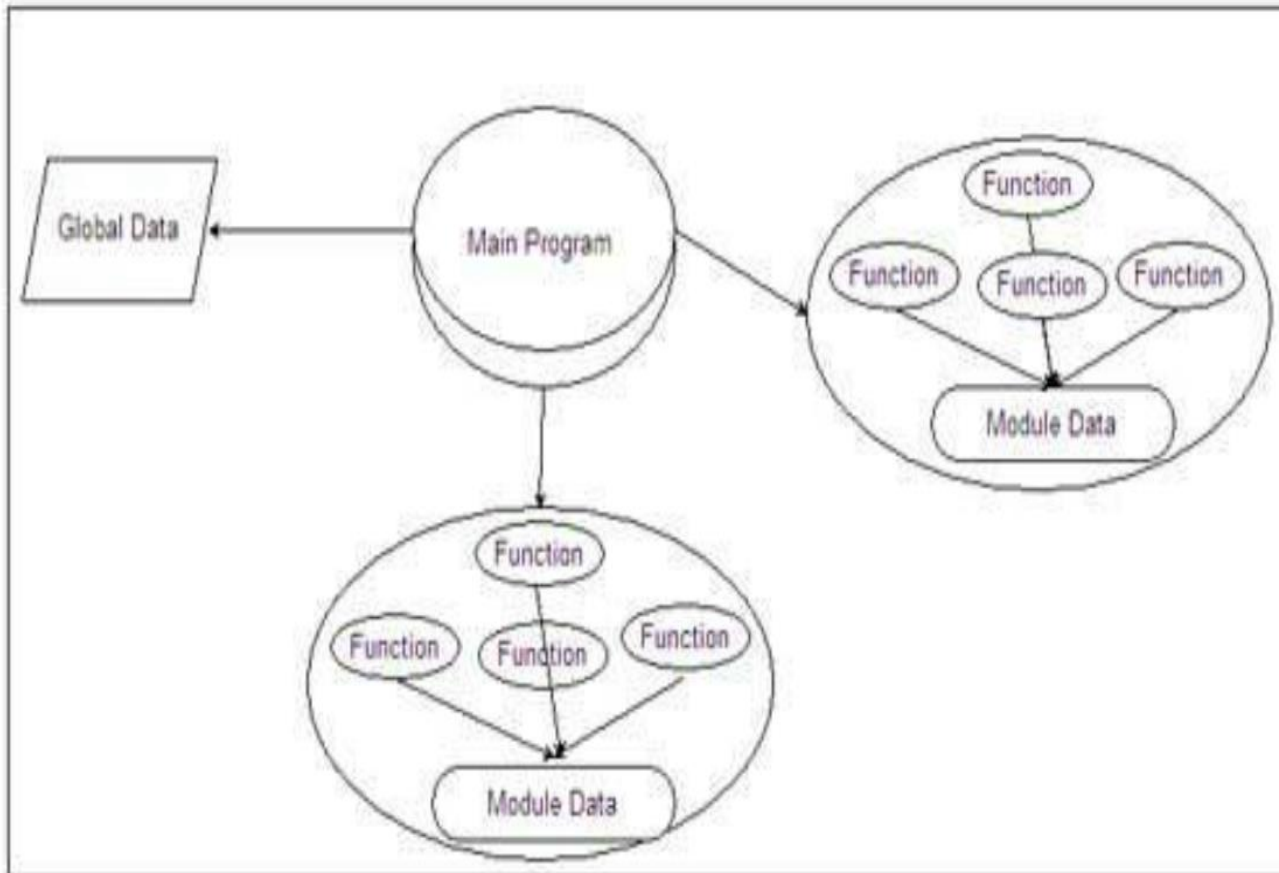
5) Design Concepts and Principles:

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, the greater will be the **effort** required to integrate the modules.

5) Design Concepts and Principles:

Modularity:



- **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.

5) Design Concepts and Principles:

Modularity:

How do we define an appropriate module of a given size? Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

- 1) **Modular decomposability:** If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.
- 2) **Modular composability.** If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.
- 3) **Modular understandability.** If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.
- 4) **Modular continuity.** If small changes to the system requirements result in changes to individual modules, rather than systemwide changes, the impact of change-induced side effects will be minimized.
- 5) **Modular protection.** If an abnormal condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

5) Design Concepts and Principles:

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**.
- **Information hiding is defined 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.

5) Design Concepts and Principles:

Information Hiding:

- Information hiding is of huge use when modifications are required during the testing and maintenance phase.
- Some of the advantages associated with information hiding are as follows:
 - 1) Leads to low coupling
 - 2) Emphasizes communication through controlled interfaces
 - 3) Decreases the probability of opposing effects
 - 4) Restricts the effects of changes in one component on others
 - 5) Results in higher quality software

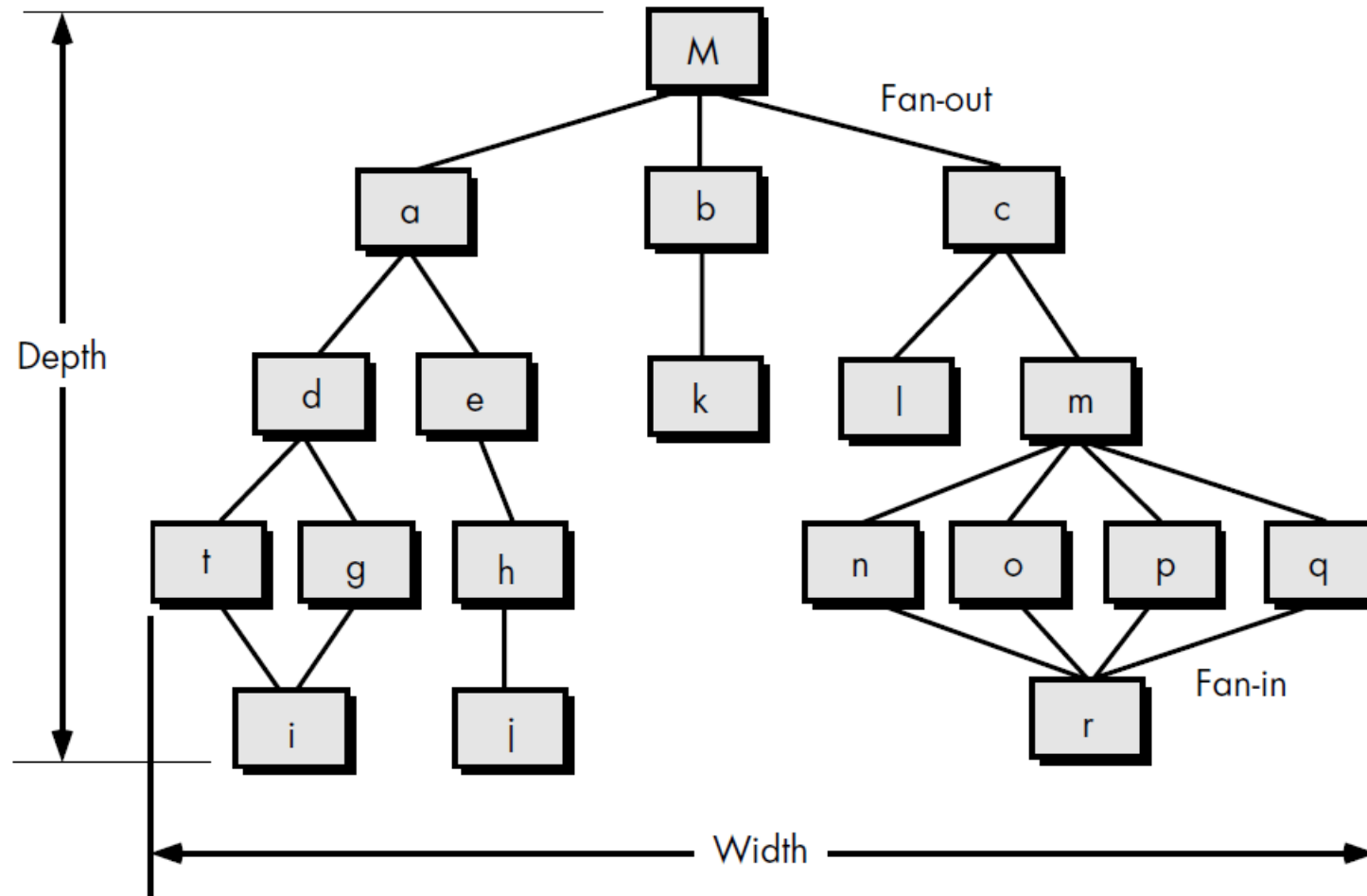
5) Design Concepts and Principles:

Control Hierarchy (Program Structure):

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

5) Design Concepts and Principles:

Control Hierarchy (Program Structure):



5) Design Concepts and Principles:

Structural Partitioning:

- When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either **horizontally** or **vertically**. In horizontal partitioning, the control modules are used to communicate between functions and execute the functions. Structural partitioning horizontally provides the following benefits.
 - 1) The testing and maintenance of software becomes easier.
 - 2) The negative impacts spread slowly.
 - 3) The software can be extended easily.
- Besides these advantages, horizontal partitioning has some **disadvantage** also. It requires to pass more data across the module interface, which makes the control flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.

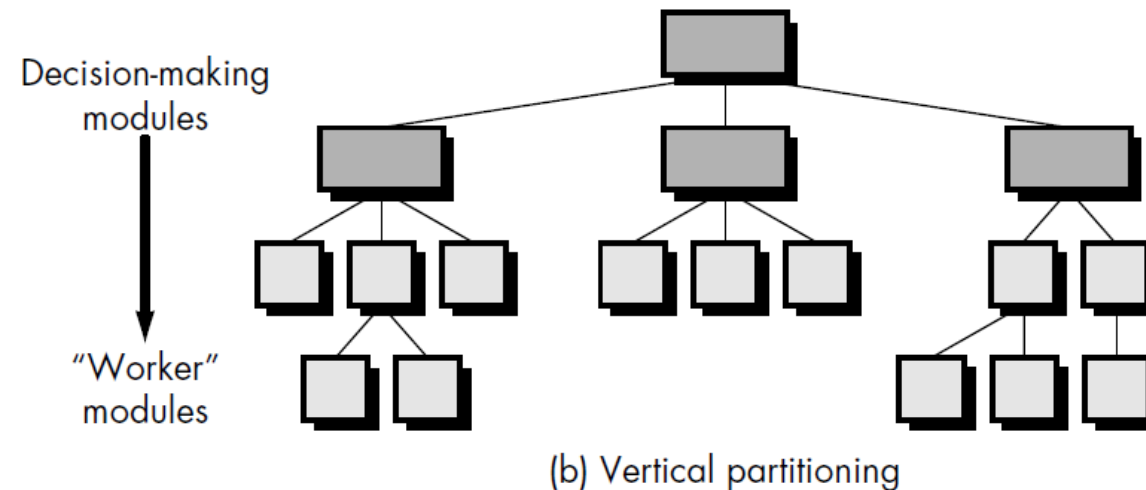
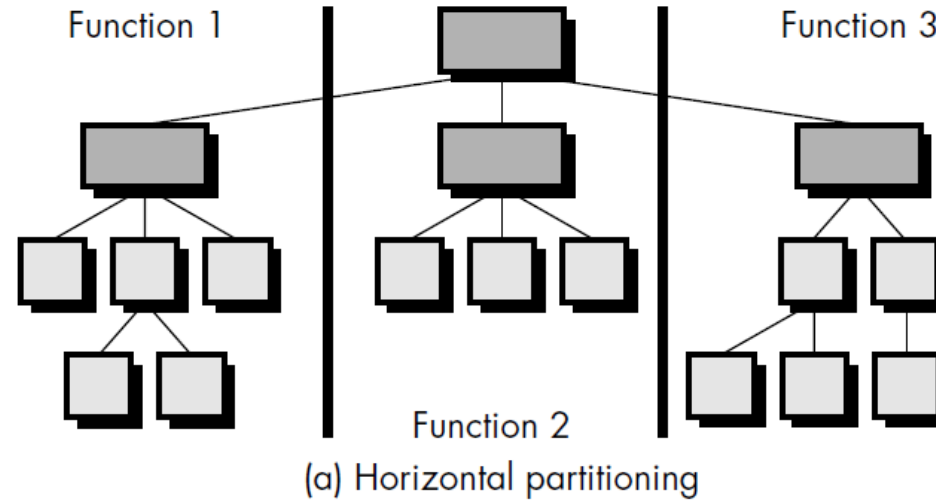
5) Design Concepts and Principles:

Structural Partitioning:

- In **vertical partitioning**, the functionality is distributed among the modules--in a top-down manner. The modules at the top level called **control modules** perform the decision-making and do little processing whereas the modules at the low level called **worker modules** perform **all input, computation and output tasks**.
- The nature of change in program structures justifies the need for vertical partitioning.
- Referring the next diagram, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a **worker module**, given to its low level in the structure, is less likely to cause the propagation of side effects.
- In general, changes to computer programs revolve around changes to input, computation or transformation, and output.
- For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable—a key quality factor.

5) Design Concepts and Principles:

Structural Partitioning:



5) Design Concepts and Principles:

Data Structure:

- Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will always affect the final procedural design, **data structure** is as important as **program structure** to the representation of software architecture.
- Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.
- The organization and complexity of a data structure are limited only by the creativity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

5) Design Concepts and Principles:

Data Structure:

- A **scalar** item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.
- When scalar items are organized as a list or contiguous group, a sequential **vector** is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.
- When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an **n-dimensional space** is created. The most common n-dimensional space is the **two-dimensional matrix**. In many programming languages, an n-dimensional space is called an **array**.

5) Design Concepts and Principles:

Functional Independence:

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

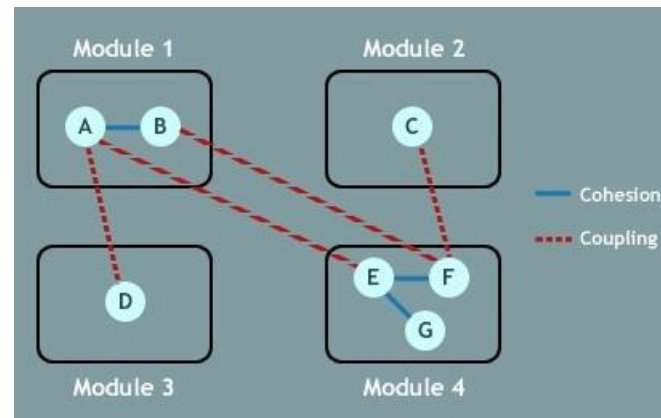
5) Design Concepts and Principles:

Need for Functional Independence:

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

5) Design Concepts and Principles:

Cohesion and coupling:



Coupling

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

5) Design Concepts and Principles:

Cohesion and coupling:

Types of Coupling:

- I. **Data Coupling:** If the dependency between the modules is based on communicating by passing only data, then the modules are said to be data coupled.
- II. **Stamp Coupling:** In stamp coupling, the complete data structure is passed from one module to another module.
- III. **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. Example- sort function that takes comparison function as an argument.
- IV. **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Examples: Ex- protocol, external file, device format, etc.
- V. **Common Coupling:** The modules have shared data such as global data structures.
- VI. **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module.

5) Design Concepts and Principles:

Cohesion and coupling:

Cohesion:

- Cohesion can be defined as the degree of the closeness of the relationship between its components. In general, it measures the relationship strength between the pieces of functionality within a given module in the software programming.

Types of Cohesion:

There are many different types of cohesion in the software engineering. Some of them are defined below:

1. **Functional Cohesion:** It is best type of cohesion, in which parts of the module are grouped because they all contribute to the module's single well defined task.
2. **Sequential Cohesion:** When the parts of modules grouped due to the output from the one part is the input to the other, and then it is known as sequential cohesion.
3. **Communication Cohesion:** In Communication Cohesion, parts of the module are grouped because they operate on the same data. For e.g. a module operating on same information records.

5) Design Concepts and Principles:

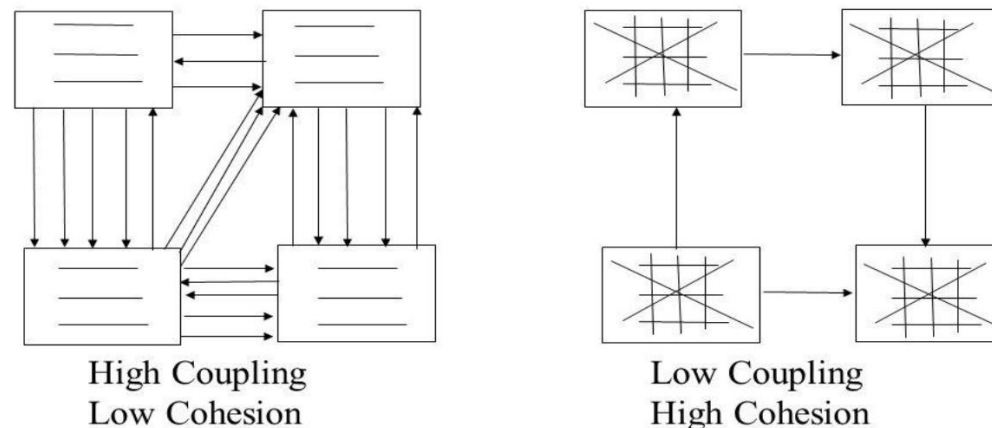
Cohesion and coupling:

Types of Cohesion:

4. **Procedural Cohesion:** In Procedural Cohesion, the parts of the module are grouped because a certain sequence of execution is followed by them.

5. **Logical Cohesion:** When the module's parts are grouped because they are categorized logically to do the same work, even though they all have different nature, it is known as Logical Cohesion.

Design Principle – Coupling and Cohesion



Assignment 2



Individual assignment

- 1) Describe the Impact of AI in Software development process
- 2) Mention advantages and disadvantages of AI in Software development process
- 3) Your answer should not exceed two pages
- 4) Write your name and registration number in your answer sheet
- 5) Submission is on **week 14**

6) Software Testing Techniques:

Software Testing Techniques

- Software testing is a critical for software quality assurance and represents the review of [specification](#), [design](#), and [code generation](#).
- The increasing visibility of software "costs" associated with a software failure are motivating forces for well-planned, thorough testing.
- It is not unusual for a software development organization to spend between 30 and 40% of total project effort on testing. In the extreme, testing of human-rated software (e.g., [flight control](#), [nuclear reactor monitoring](#)) can cost three to five times as much for testing.
- Software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test case design techniques. Software requirements are exercised using "black box" test case design techniques. In both cases, the intent is to find the maximum number of errors with the minimum amount of effort and time.

6) Software Testing Techniques:

Software Testing Techniques

- A set of test cases to exercise both internal logic and external requirements is designed and documented, **expected results are defined**, and actual **results are recorded**.
- When you begin testing, change your point of view. Try hard to “break” the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness.
- In testing, the engineer creates a series of test cases that are intended to "demolish" the software that has been built. In fact, testing is the one step in the software process that could be viewed as professional destructive to software rather than constructive.

Testing Objectives

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an undiscovered error.
3. A successful test is one that uncovers an undiscovered error.

6) Software Testing Techniques:

Testing Principles:

1. All tests should be traceable to customer requirements. "most severe defects are those that cause the program to fail to meet its requirements"
2. Tests should be planned long before testing begins. "all tests can be planned and designed before any code has been generated"
3. Testing should begin "in the small" and progress toward testing "in the large." "The first tests focus on individual components then as progresses, focus shifts to find errors in integrated components and ultimately for the entire system"
4. Exhaustive testing is not possible. "It is impossible to execute every combination of paths during testing"
5. To be most effective, testing should be conducted by an independent third party. "The software engineer who created the system is not the best person to conduct the testing"

6) Software Testing Techniques:

Testability:

- In ideal circumstances, a software engineer designs a computer program, a system, or a product with “testability” in mind. This enables the individuals charged with testing to design effective test cases more easily.
- **Software testability** is simply how easily (a computer program) can be tested.
- The following checklist provides a set of characteristics that lead to testable software:
 - 1) **Operability**. "The better it works, the more efficiently it can be tested."
 - 2) **Observability**. "What you see is what you test."
 - 3) **Controllability**. "The better we can control the software, the more the testing can be automated and optimized."
 - 4) **Decomposability**. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
 - 5) **Simplicity**. "The less there is to test, the more quickly we can test it."
 - 6) **Stability**. "The fewer the changes, the fewer the disruptions to testing."
 - 7) **Understandability**. "The more information(documentation) we have, the smarter we will test."

6) Software Testing Techniques:

Test Case Design:

- The design of tests for software and other engineered products can be as challenging as the initial design of the product itself.
- Any engineered product can be tested in one of two ways:
 1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
 2. Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.
- The first test approach is called "**black-box testing**" and the second, "**white-box testing**".

6) Software Testing Techniques:

Black-box Testing:

Black-box testing, also called [behavioral testing](#), focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain.

6) Software Testing Techniques:

White-box Testing:

White-box testing, sometimes called [glass-box testing](#), is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a [module](#) have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

White-box testing is described as "testing in the small." Its implication is that it is applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens our focus and might be called "testing in the large." Black-box tests are designed to validate functional requirements without regard to the internal workings of a program.

6) References:

- 1) Software engineering: a practitioner's approach / Roger S. Pressman.—5th ed. 2001
- 2) "Introduction to Software Engineering" All right reserved 2011-2023 copyright © computer-pdf.com v5
- 3) Various Introduction to Software Engineering notes

End!