

SOFTWARE ENGINEERING (COSC2042)

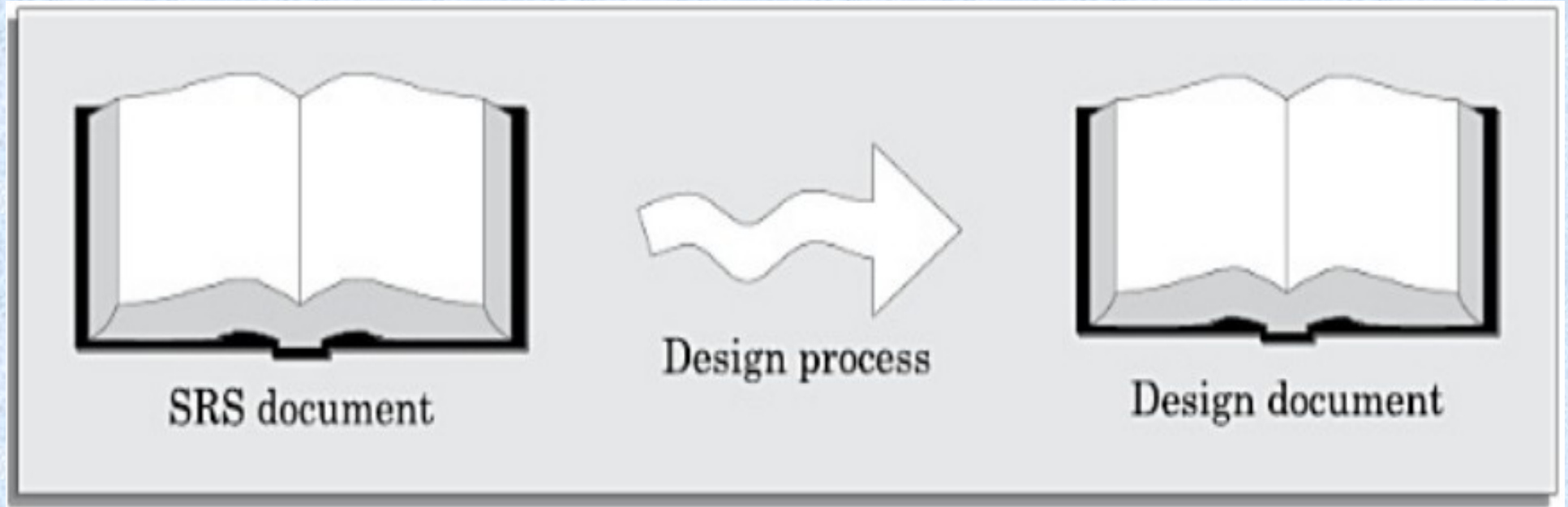
Chapter - 5: Software Design

1

INTRODUCTION

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms.
- The output of this process can directly be used into implementation in programming languages.
- Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from **problem domain** to **solution domain**. It tries to specify how to fulfill the requirements mentioned in SRS.

SOFTWARE DESIGN PROCESS



SOFTWARE DESIGN LEVELS

- **High-Level Design:** High-level design (HLD) is the initial phase of design where the overall structure and organization of the system are conceptualized at a high level.
- It focuses on defining the major components or modules of the system, their interactions, and the relationships between them. It addresses system-wide concerns and provides a blueprint for the architecture.
- The abstraction level is high, and the design may involve block diagrams, system flowcharts, or high-level data models.
- The primary output of high-level design is often represented through architectural diagrams, such as block diagrams or system diagrams.

Cont...

- **Architectural Design:** Architectural design is a more detailed phase that follows high-level design. It involves specifying the overall structure of the system, including subsystems, modules, and their interactions.
- The focus is on defining the key architectural components, their responsibilities, interfaces, and the overall system behavior. It addresses issues related to scalability, maintainability, and performance.
- While more detailed than high-level design, it still maintains a level of abstraction.
- The output includes architectural diagrams, component diagrams, and sometimes interaction diagrams that illustrate the relationships between major components.

Cont...

- **Detailed Design:** Detailed design is the phase where the low-level design of individual modules or components is carried out. It provides a blueprint for implementing each module.
- The focus is on specifying algorithms, data structures, and implementation details for each module. It addresses how each component will fulfill its responsibilities as defined in the architectural design.
- The level of abstraction is much lower compared to high-level and architectural design. Code may be pseudocode or detailed specifications.
- The output includes detailed specifications for each module, data structures, algorithms, and sometimes even code snippets.

MODULARIZATION

- **Modularization** is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently.
- These modules may work as basic constructs for the entire software.
- Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

ADVANTAGE OF MODULARIZATION

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again.
- Concurrent execution can be made possible
- Desired from security aspect

CONCURRENCY

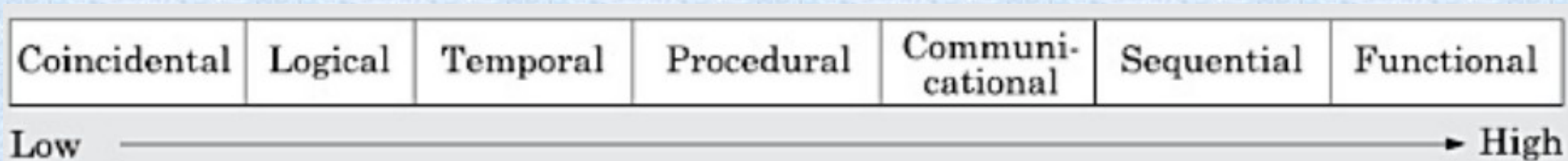
- In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel.
- In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.
- It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

COUPLING AND COHESION

- When a software program is modularized, its tasks are divided into several modules based on some characteristics.
- As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together.
- There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

COHESION

- Cohesion is a measure that defines the degree of intra-dependability within elements of a module.
- The greater the cohesion, the better is the program design.
- There are **seven** types of cohesion.
- As shown in the figure below the cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible.



COHESION

- Let's describe each of them in detail:
 - **Co-incident cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
 - **Logical cohesion** - A module is said to be **logically cohesive**, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
 - As an **example** of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

COHESION

- **Temporal Cohesion** - When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
- As an **example**, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.

COHESION

- **Procedural cohesion** - A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
 - **For example**, Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print-bill(), place-order-on-vendor(), update-inventory(), and logout () all do different things and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.

COHESION

- Communicational cohesion - A module is said to have communicational cohesion, if all functions of the module **refer to** or **update** the **same data structure**.
 - As an **example** of procedural cohesion, consider a module named student in which the different functions in the module such as **admitStudent**, **enterMarks**, **printGradeSheet**, etc. access and manipulate data stored in an array named **studentRecords** defined within the module.
- Sequential cohesion - When elements of module are grouped because the **output of one element serves as input to another** and so on, it is called sequential cohesion.
 - For **example**, In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions `create-order()`, `check-item-availability()`, `place-order-on-vendor()` are placed in a single module, then the module would exhibit sequential cohesion.

COHESION

- Functional cohesion - A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task.
- For **example**, a module containing all the functions required to **manage employees' pay-roll** displays functional cohesion. In this case, all the functions of the module (e.g. , **computeOvertime()**, **computeWorkHours()**, **computeDeductions()**, etc.) work together to **generate the payslips** of the employees.

COUPLING

- Coupling is a measure that defines **the level of inter-dependability among modules** of a program.
- It tells at what level the modules interfere and interact with each other.
- The **lower the coupling**, the **better the program**. There are five levels of coupling. As shown below data coupling has lowest coupling and content coupling has sever coupling.
- There are five levels of coupling, namely:
 - **Content coupling** - When a module can **directly access** or **modify** or **refer to the content of another module**, it is called content level coupling.
 - **Common coupling** - When multiple modules **have read and write access** to some **global data**, it is called common or global coupling.
 - **Control coupling** - Two modules are called control-coupled **if one of them decides the function of the other module or changes its flow of execution**.



COUPLING

- **Stamp coupling-** When multiple modules **share common data structure** and **work on different part of it**, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other **by means of passing data** (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.
- Ideally, no coupling is considered to be the best.

SOFTWARE DESIGN STRATEGIES

- Software design is a process to **conceptualize** the **software requirements** into **software implementation**.
- Software design takes the user requirements as challenges and tries to find optimum solution.
- While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.
- There are multiple variants of software design. Let us study them briefly:
 - Structured design
 - Function oriented design
 - Object oriented design

STRUCTURED DESIGN

- Structured design is a conceptualization of problem into several well-organized elements of solution.
- It is basically concerned with the **solution design**.
- **Benefit** of structured design is, it **gives better understanding** of how the problem is being solved.
- Structured design also makes **it simpler for designer** to concentrate on the problem more accurately.
- Structured design is mostly based on ‘**divide and conquer**’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
- The small pieces of problem are solved by means of solution modules.
- Structured design emphasis that these modules be well organized in order to achieve precise solution.

FUNCTION ORIENTED DESIGN

- In function-oriented design, the system is comprised of many smaller sub-systems known as **functions**. These functions are capable of performing significant task in the system.
- The system is considered as top view of all functions.
- 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.

OBJECT ORIENTED DESIGN

- Object oriented design works around the **entities** and their **characteristics** instead of functions involved in the software system. This design strategy focuses on entities and its characteristics.
- The whole concept of software solution revolves around the engaged entities.
- Let us see the important concepts of Object Oriented Design:
 - **Objects** - 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.
 - **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.

OBJECT ORIENTED DESIGN

- In the solution design, **attributes** are stored as **variables** and **functionalities** are defined by means of **methods** or **procedures**.
- **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. This is called **information hiding**.
- **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.

OBJECT ORIENTED DESIGN

- 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.

SOFTWARE DESIGN APPROACHES

- There are two generic approaches for software designing: top-down design and bottom-up design.
- **Top-Down Design:**
 - **Approach:** Start with a high-level overview of the system and progressively refine the design by breaking it down into more detailed sub-modules.
 - **Benefits:** Provides a clear understanding of the overall structure before diving into details. Allows for incremental development.
- **Bottom-Up Design:**
 - **Approach:** Begin by designing the detailed components and gradually combine them to form larger subsystems and the complete system.
 - **Benefits:** Focused attention on small components, making it easier to test and debug. Encourages the reuse of well-designed components.



Thank you!!