W5 Summary

Name: Prateek P

- Content taken from **Fundamentals of Software Engineering by Rajib Mall, IIT KGP**

Aspects of Design

A design solution should be modular and layered for better understanding and management of complexity. Two key aspects of design—**Modularity** and **Layered Design**—play a crucial role in achieving this goal.

Modularity refers to decomposing a problem into modules. A modular design has high **cohesion** and low **inter-module couplings**. High cohesion means each module works towards a single objective, while low coupling means the modules have little or no interaction with each other. This decomposition, based on the principle of "divide and conquer," allows each module to be understood and worked on independently.

Layered Design is another important concept, where modules are organized in layers. The hierarchical structure ensures that each module only interacts with modules in the layer directly beneath it, forming a tree-like diagram. This layered approach helps maintain control abstraction, making the system easier to manage and debug.

Cohesion

Cohesion measures how well the functions within a module work together to achieve a single objective. Just like a well-structured speech focuses on one theme, cohesive modules ensure that all functions cooperate towards the same task. The levels of cohesion, from lowest to highest, include:

1. **Coincidental Cohesion**: Functions are unrelated and randomly grouped, like managing library books and handling staff leave in the same module.
2. **Logical Cohesion**: Functions perform similar operations, such as different types of data handling (e.g., error handling or input/output).
3. **Temporal Cohesion**: Functions execute within the same time span, such as system boot-up tasks.
4. **Procedural Cohesion**: Functions execute one after another but perform unrelated tasks.
5. **Communicational Cohesion**: Functions share and manipulate the same data structure, such as a student records module.
6. **Sequential Cohesion**: Functions depend on the output of the previous one, as seen in some processing pipelines.

7. **Functional Cohesion**: All functions work towards completing a single, unified task, like an employee payroll system that computes hours, deductions, and overtime.

Coupling

Coupling refers to the degree of interdependence between modules. Lower coupling is preferable as it reduces complexity. The types of coupling, from least to most dependent, are:

1. **Data Coupling**: Modules share simple data items like integers or floats.
2. **Stamp Coupling**: Modules share complex data structures.
3. **Control Coupling**: A module controls the flow of another module based on shared data.
4. **Common Coupling**: Modules share global variables.
5. **Content Coupling**: One module directly accesses or alters the contents of another, which should be avoided.

Functional Independence is achieved when modules perform single tasks and require minimal interaction with others. This promotes **error isolation** (errors in one module don't affect others), **reuse** (modules can be easily reused), and **understandability** (modules are simpler to comprehend individually).

Hierarchical Design

A good design also emphasizes **control hierarchy**, which determines the organization of modules. Characteristics like **fan-in** (how many modules call a given module) and **fan-out** (how many modules a given module controls) help balance control and modularity. High fan-in encourages code reuse, while **layered arrangement** in hierarchical designs maintains control abstraction by restricting module calls to lower layers.

Design Approaches

Design approaches fall into two categories: **Function-Oriented Design** and **Object-Oriented Design**.

Function-Oriented Design

In function-oriented design, systems are broken down into a hierarchy of functions using **top-down decomposition**. A system is viewed as a set of high-level functions that are successively refined into more detailed subfunctions. For example, creating a library member involves assigning a membership number, creating a record, and printing a bill, each broken down into further subfunctions. Function-oriented design often uses **centralized system state**, where data is stored globally and shared across functions.

Object-Oriented Design (OOD)

In OOD, the focus is on real-world entities or **objects**, each containing data and methods. Objects encapsulate data and control access through methods, decentralizing the system state. This abstraction improves modularity by localizing data to objects, making the system more modular and cohesive. Objects in OOD communicate by **message passing**, where one object can discover the state of another by sending messages, rather than relying on shared global data.

In OOD, **data abstraction** hides the details of how data is stored or manipulated inside objects. This allows for greater flexibility and reusability. Objects are treated as instances of **abstract data types (ADTs)**, which ensure **encapsulation** of data, so errors can be traced to specific methods responsible for manipulating that data.

Comparison of Function-Oriented and Object-Oriented Design

- In **Function-Oriented Design**, the primary abstraction is services or functions available to users, such as issuing a book or printing a report. The system state is centralized, stored globally, and shared across functions.
- In **Object-Oriented Design**, the abstraction is based on real-world entities, like members or books. The system state is decentralized, with data distributed among different objects. Functions are grouped around the data they manipulate, and objects interact through message passing.

Thus, object-oriented design offers better modularity, cohesion, and reusability, while function-oriented design is more focused on the top-down breakdown of tasks. Both approaches have their own use cases, but OOD, with its emphasis on encapsulation, abstraction, and independence, offers a more scalable and maintainable solution in complex systems.