Chapter 1: Introduction

1 Software Design

Design

In software engineering, design refers to the phase that comes before the actual coding. It involves steps like creating algorithms, breaking down a complex system into manageable parts (modules), and brainstorming the general ideas and logic for each part, including how they will communicate with each other. A well-crafted design is crucial, as it not only makes the implementation and testing phases smoother but also ensures that maintenance is more manageable. Proper maintenance is key to making the software last for a long time, so it's important to focus on sustainable design practices.

2 Maintenance

Maintenance

Maintenance is the final phase in the lifecycle of any software. It involves fixing bugs, adding, removing, or modifying features. A good design plays a crucial role in making maintenance more manageable and efficient.

Note

A good designer knows how to make the right choices and weigh the pros and cons. In real-life applications, designers face very complex systems, and it's not always possible to have all the advantages.

3 Modularity (Decomposition)

Modularity

Modularity divides a complex system into smaller parts called modules:

1. **Architectural Design**: Defines the modules and how they will communicate with each other.
2. **Unit Design**: Defines each module's elements independently of the others.

4 Evaluation Criteria

Criteria for a Good Decomposition

Many decompositions (modularities) might be proposed for the same system, and the criteria for a good decomposition are:

- High Cohesion
- Low Coupling

4.1 Cohesion

4.1.1 Definition

Definition

Cohesion refers to the degree of structuring and affinity between the elements of a module. It essentially describes how tightly related the goals and tasks of these elements are. There are several types of cohesion, which we will list from weakest to strongest. A strong, high cohesion is desirable.

4.1.2 Why High Cohesion

High Cohesion

The stronger the cohesion, the more tightly related the elements of the module are, making the module adhere to the Single Responsibility Principle. This results in easier maintenance because when updating a specific feature, we can easily pinpoint which module is related to it. Since each module has a single responsibility, changes are more localized and manageable.

4.1.3 Types Of Cohesion

Random Cohesion (Coincidental Cohesion)

Random

The elements of a module are grouped together randomly with no meaningful relationship. This is the weakest form of cohesion and should always be avoided.

Logical Cohesion

Logical

Groups elements of a module that perform logically similar tasks.

Temporal Cohesion

Temporal

Groups elements of a module that execute within the same time phase.

Procedural Cohesion

Procedural

Groups elements of a module that execute in a well-defined order (collaboration of procedures).

Communicational Cohesion

Communicational

Groups elements of a module that operate on the same set of data.

Sequential Cohesion

Sequential

Groups elements of a module so that the output of one element serves as the input for another.

Functional Cohesion

Functional

All elements contribute to a single, well-defined task or are indispensable to its execution.

Note

- Random cohesion must be avoided. If it appears in a test or exam, it results in an automatic 0. We aim for functional cohesion, as the systems studied in exercises are simple.
- In real-life applications, achieving pure functional cohesion is not always possible, and hybrid cohesion may be necessary.

4.2 Coupling

4.2.1 Definition

Definition

Coupling refers to the degree of dependency (Interconnection) between modules. We aim for low, weak coupling to promote modularity and maintainability,it has many types we will list them from strongest to weakest.

Note

No coupling is impossible, as any system requires minimum communication between modules.

4.2.2 Why Low Coupling

Low Coupling

We want low coupling so that modules are loosely coupled to:

- **Failure Prevention:** If several modules depend on a central module, and it fails, the whole system crashes.
- **Updating Modules:** Having low coupling means that when updating a module, it will impact fewer modules, making the process easier.

4.3 Types Of Coupling

4.3.1 Content Coupling

Content

One module directly accesses or modifies the internal content of another module. This is the strongest type of coupling and should be avoided.

4.3.2 Common Coupling

Common

Modules that share global data.

4.3.3 External Coupling

External

Modules that communicate with each other through external entities.

4.3.4 Control Coupling

Control

One module controls the behavior of another by passing control parameters.

4.3.5 Stamp Coupling

Stamp

Modules that share complete complexe data structures (non-primitive : collection, array , struct , object) , but not all the elements are used by each module..

4.3.6 Data Coupling

Data

Modules share only essential simple data (primitive data) through parameters or arguments.

4.4 Relation Between Cohesion & Coupling

Relation

- **High cohesion leads to low coupling:** A highly cohesive module depends less on other modules and more on its internal elements, resulting in low coupling.
- **Low coupling leads to high cohesion:** Low coupling means that modules are not highly dependent on each other, so the elements within each module need to be sufficient to perform a task, leading to high cohesion.

$$\boxed{\text{High Cohesion} \iff \text{Low Coupling}}$$

5 Scalability

Definition

As the number of users, data size, and server requests grow, our software must continue running smoothly and adapt to exponential growth without slowing down or crashing. To achieve this, we need to ensure that our software is scalable.

5.1 Principles of Scalability

Principles

- **Scalable Database**: Ensures efficient data storage and retrieval as demand increases.
- **Distributed Architecture**: Spreads the workload across multiple servers to prevent bottlenecks.
- **Asynchronous Processes**: Improves efficiency by allowing tasks to run in the background without blocking other operations, enabling the system to handle more requests simultaneously.
- **Microservices**: Breaks applications into smaller, independent services deployed in different servers for better scalability.
- **Load Balancing**: Distributes traffic evenly to prevent server overload.
- **Auto-scaling**: Adjusts resources dynamically based on demand.
- **Caching**: Stores frequently accessed data to reduce load and improve speed.

5.2 Types of Scalability

5.2.1 Horizontal (Flat) Scalability

Horizontal

Scaling by adding more servers to distribute the load. For example, deploying additional servers in different regions to improve performance.

5.2.2 Vertical Scalability

Vertical

Scaling by upgrading the same server with better hardware. For example, increasing RAM to enhance caching performance.