**Design principles**

Design means how. Having good design means it has to be deliver faster,manage changes, deal complexities.

Bad design is caused due to rigidity,viscosity, fragility,immobality.

**Rigidity :**

It refers to the difficulty of making changes to a system. A rigid system is one where making modifications, even small ones, is challenging and may require extensive effort.

**Viscosity:**

viscosity refers to the resistance a system exhibits against changes or improvements. It's a measure of how easy or difficult it is to maintain and evolve the system in a desired direction.

**Fragility:**

Fragility refers to the system's susceptibility to breaking or failing when changes are made. A fragile system is one where a small modification in one part may cause unexpected issues or failures in other parts.

**Immobility:**

Immobility occurs when it is challenging to reuse or move components of the system to other contexts. It implies a lack of flexibility or portability.

Good design requires high cohesion and low coupling

**Differences between cohesion and coupling:**

Key Differences:

**Focus:**

Cohesion is concerned with the internal relationships within a module.

Coupling is concerned with the external relationships and dependencies between modules.

**Goal:**

The goal of cohesion is to have modules that are internally focused and perform a specific task well.

The goal of coupling is to have modules that are loosely connected, reducing dependencies and making the system more flexible.

**Impact of Change:**

High cohesion makes it easier to make changes within a module without affecting other parts of the system.

Loose coupling minimizes the impact of changes in one module on other modules.

Five principles for good designs:

**Solid principle:**

It is introduced by Robert C. Martin

**Single Responsibility Principle (SRP):**

A class should have only one reason to change, meaning it should have only one responsibility or job.

Impact: Encourages modular design, making classes more focused, easier to understand, and less prone to modifications due to multiple reasons.

**Open/Closed Principle (OCP):**

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In other words, you should be able to add new functionality without altering existing code.

Impact: Promotes the use of abstraction and polymorphism to extend system behavior without changing existing code, enhancing maintainability and reducing the risk of introducing bugs.

**Liskov Substitution Principle (LSP):**

Subtypes must be substitutable for their base types without altering the correctness of the program. Inheritance should not break the functionality of the base class.

Impact: Ensures that derived classes adhere to the expected behavior of base classes, promoting consistency and avoiding unexpected behavior when using polymorphism.

**Interface Segregation Principle (ISP):**

A class should not be forced to implement interfaces it does not use. Clients should not be forced to depend on interfaces they do not use.

Impact: Encourages the creation of smaller, specific interfaces, preventing classes from being burdened with unnecessary methods and reducing dependencies between components.

**Dependency Inversion Principle (DIP):**

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

Impact: Promotes the use of abstractions (interfaces or abstract classes) to decouple high-level and low-level modules, making the system more flexible, scalable, and easier to test.

**KISS-keep it simple stupid**

KISS is based on the idea that simple code is easier to document, read, maintain, and debug than complicated code. KISS emphasizes the importance of keeping code simple and meaningful.

**DRY-Don’t repeat Yourself**

This principle suggests that developers should strive to write code that can be reused and avoid writing the same code multiple times.

DRY is a principle that emphasizes on avoidance of code duplication.

**YAGNI-You Aint Gonna need it**

This principle suggests that developers should not write code that is not needed or might not need.

This principle helps developers avoid premature optimization and unnecessary code.

**Dependency injection:**

Dependency Injection (DI) is a software design pattern that allows us to develop loosely coupled code. •DI is a great way to reduce tight coupling between software components. •DI also enables us to better manage future changes and other complexity in our software. •The purpose of DI is to make code maintainable.

**Constructor Injection**

We can inject the depedency to the main component directly in the constructor when the component is instantiated. In this case the dependency should be already instantiated when the constructor of the main component is invoked.

**Property Injection:-**

This type of dependency injection consists in passing the dependency through a property. It allows a lazy loading initialization of the dependency, in case the dependency is not instantiated when main component is instantiated. Further more, it makes the code less coupled, since the constructor of the main component does not have to know anything about the dependency.

**Interface Injection: -**

Interface Injection is a design approach related to Dependency Injection (DI) that emphasizes injecting dependencies through interfaces. While Interface Injection provides flexibility and adherence to DIP, it requires more setup compared to other injection methods like Constructor Injection. The choice of injection type depends on the specific needs and constraints of the project.