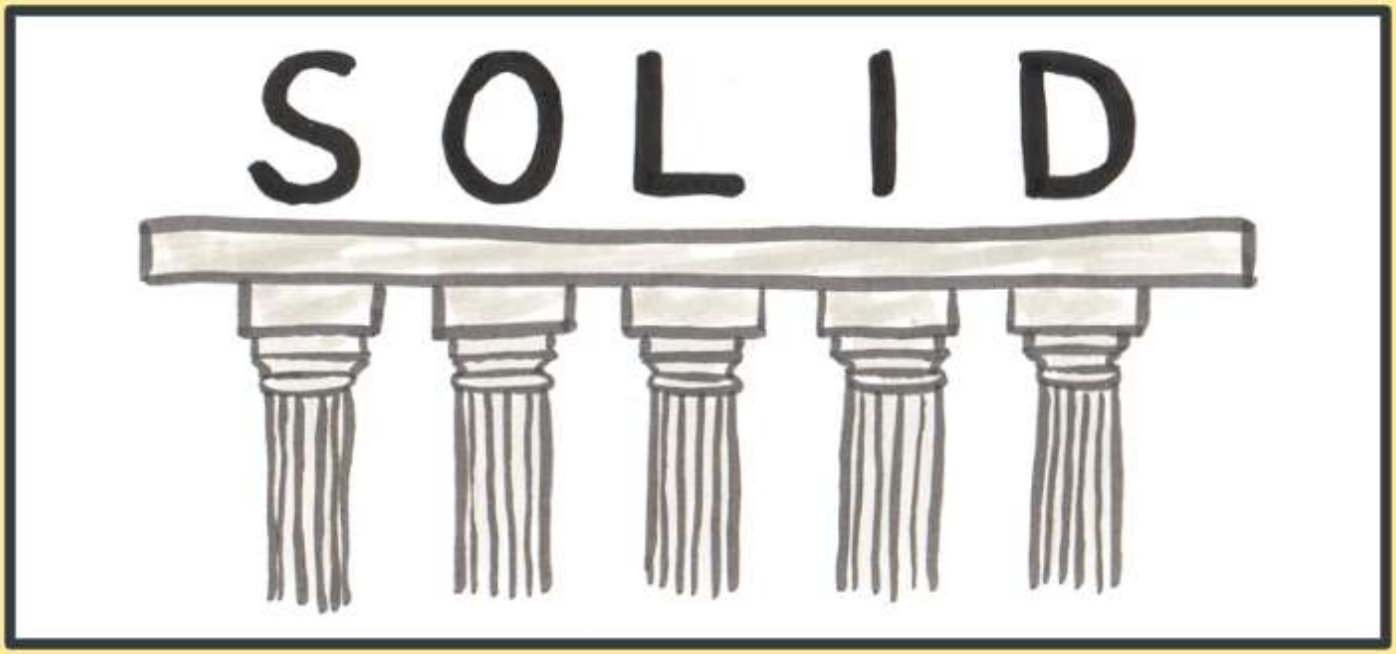
Complete Notes On



LOW LEVEL DESIGN

1. What is A Good Coder / Quality Code?

- A good coder produces quality code that is not only functional but also maintainable, scalable, and robust.
- Quality code adheres to coding standards, is well-documented, and follows best practices.
- It should be easy to understand, modify, and extend without introducing bugs or unexpected behaviour.

A good coder produces quality code that possesses the following characteristics:

1. **Readable:** Code is easy to understand and maintain.
2. **Rememberable Code:** Comments are used judiciously to explain complex logic or algorithms.
3. **Extensible:** Code can be easily extended without major modifications.
4. **Flexible:** Code is adaptable to changes in requirements or environment.
5. **Stable:** Exception handling is implemented effectively to ensure stability.
6. **Modular:** Code is organized into logical modules, promoting reusability and maintainability.
7. **Secure:** Security measures are implemented to protect against vulnerabilities and threats.
8. **Correct:** Code functions as intended and produces expected outcomes.

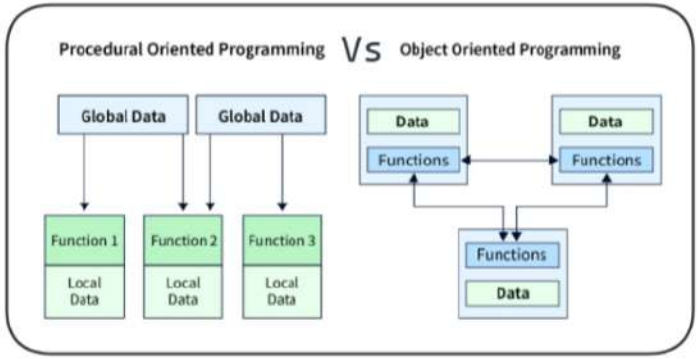
2. Functional Vs Object Oriented Programming

Functional Programming **(FP)** and Object-Oriented Programming **(OOP)** are two paradigms used in software development.

Functional Programming: (TREE) Focuses on the evaluation of expressions, avoiding changing-state and mutable data. Functions are treated as first-class citizens, enabling higher-order functions and immutability.

Object-Oriented Programming: (LEGO) Organizes software design around objects, which encapsulate data and behaviour. Objects interact with each other through methods and messaging.

Both paradigms have their strengths and weaknesses, and the choice between them depends on the specific requirements of the project.



3. What is SOLID Principles?

*The **SOLID** principles are a set of five fundamental design principles that were introduced by **Robert C. Martin (Uncle Bob)**, named by Michael Feathers.*

Why SOLID Principles? (Good Coder)

- *To make code more maintainable, easier to understand, easy to use.*
- *To make it easier to quickly extend the system with new functionality without breaking the existing ones.*
- *To make the code easier to read and understand, thus spend less time figuring out what it does and more time actually developing the solution. (Time Saving)*

The SOLID principles are:

1. *Single Responsibility Principle **(SRP)***
2. *Open-Closed Principle **(OCP)***
3. *LISKOV Substitution Principle **(LSP)***
4. *Interface Segregation Principle **(ISP)***
5. *Dependency Inversion Principle **(DIP)***

	Principle	Description
S	**Single Responsibility Principle**	Each class should be responsible for a single part or functionality of the system
O	**Open-Closed Principle**	Software components should be open for extension, but not for modification.
L	**Liskov Substitution Principle**	Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
I	**Interface Segregation Principle**	No client should be forced to depend on methods that it does not use.
D	**Dependency Inversion Principle**	High-level modules should not depend on low-level modules, both should depend on abstractions.

1. What is single Responsibility Principle (SRP)

1. A class should have one, and only one reason to change.
   This means that a class should only have one job or responsibility.
2. A class should only be responsible for one thing.
3. There's a place for everything and everything in its place.
4. Find one reason to change and take everything else out of the class.
5. Importance: Following `**SRP**` makes your code more modular, easier to understand, maintain, and extend.
   It helps in isolating functionalities, making debugging and testing more straightforward.

In One Statement
The Single Responsibility Principle states that a class should have only one reason to change, meaning it should have only one job or responsibility. This promotes modularization and makes the code easier to understand and maintain.

Key Idea:
A class should do only one thing, and it should do it well.

Real-Time Examples:
1. Think of a chef who only focuses on cooking, not managing the restaurant or delivering food.
2. In auth system, user only focuses on login, not sending email or verify email
3. In a household setting, the book container exclusively stores books, the food bucket is designated for food items only, and the tool box is specifically reserved for tools, ensuring each container serves its intended purpose without mixing items.

How can Single Responsibility Principle be applied?

Visit GitHub:
@BCAPATHSHALA

Practical Coding Examples in Java #1
Practical Coding Examples in Java #2
Practical Coding Examples in Java #3

1. What is Open-Closed Principle (OCP)

1. An entity `(e.g., classes, modules, functions, etc.)` should be open for extension but closed for modification.
   This means you should be able to add new functionality without changing the existing code.
2. Extend functionality by adding new code instead of changing existing code.
3. **Goal:** Get to a point where you can never break the core of your system.
4. **Importance:** `OCP` encourages a more stable and resilient codebase. It promotes the use of `interfaces` and `abstract classes` to allow for **behaviours to be extended without modifying existing code.**
5. Writing code structure in such a way new functionality can be added by adding new code not by modifying existing code.
6. **OPEN** for extending and **CLOSE** for modification

In One Statement:
The Open-Closed Principle states that classes should be open for extension but closed for modification. This means that the behaviour of a class should be extendable without modifying its source code.

Key Idea:
Once a class is written, it should be closed for modifications but open for extensions.

Real-Time Examples:
1. Your smartphone – you don't open it up to add features; you just download apps to extend its capabilities.

How can Open-Close Principle be applied?

Visit GitHub: @BCAPATHSHALA

Practical Coding Examples in Java #1
Practical Coding Examples in Java #2
Practical Coding Examples in Java #3
Much more about ***Open-Close Principle***

1. LISKOV Substitution Principle (LSP)

1. Any `**derived**` class should be able to substitute its `**parent**` class without the consumer knowing it.
2. Every part of the code should get the expected result no matter what instance of a class you send to it, given it implements the same interface.
3. If a function takes a `Base` class as parameter then, this code should work for all the `Derived` classes.
4. **LSP** insures that the good application i.e., built using **abstraction** does not break.
5. It states that the objects of a `subclass` should behave the same way as the objects of the `superclass`, such that they are **replaceable**.
6. **Key:** `**Child**` class should be able to do what a `**parent**` class can.
7. **Goal**: The goal of **LSP** is to ensure that a `**subclass**` can stand in for its `**superclass**`. This principle helps in maintaining the correctness of the program when objects of a superclass are replaced with objects of a subclass.

In One Statement
*The **LISKOV Substitution Principle** states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.*

Key Idea
You should be able to use any `**subclass**` where you use its `**parent**` class.

Real-Time Examples
You have a remote control that works for all types of `**TVs**`, regardless of the `**brand**`. Where,
*- **Brand:** as a Parent class*
*- **Remote Control:** as a feature of Brand*
*- **TVs:** as a Child class*

How can LISKOV Substitution Principle be applied?

Visit GitHub:
@BCAPATHSHALA

Practical Coding Examples in Java #1
Practical Coding Examples in Java #2
Practical Coding Examples in Java #3
*Much more about **LISKOV Substitution Principle***

1. What is Interface Segregation Principle (ISP)

1. The **Interface Segregation Principle (ISP)** is a design principle that does not recommend having methods that an interface would not use and require.
2. Therefore, it goes against having fat interfaces in classes and prefers having small interfaces with a group of methods, each serving a particular purpose.
3. To comply with the Interface Segregation Principle **(ISP)**, it's important to design interfaces that are tailored to specific client needs instead of creating broad, all-purpose interfaces.
4. Do not build one pet interface **(large interface)** make smaller and specific ones.

In One Statement
This principle encourages the creation of small, more client-specific interfaces.

Key Idea
ISP: *Create a different interface for each responsibility; don't group unrelated behaviour into one interface.*
LSP: *Requires you to ensure that all child classes have the same behaviour as the parent class.*

Real-Time Examples
You sign up for a music streaming service and only choose the genres you like, not all available genres.

How can Interface Segregation Principle be applied?

Visit GitHub:
@BCAPATHSHALA

Practical Coding Examples in Java #1
Practical Coding Examples in Java #2
Practical Coding Examples in Java #3
Much more about ***Interface Segregation Principle***

What is Dependency Inversion Principle (DIP)

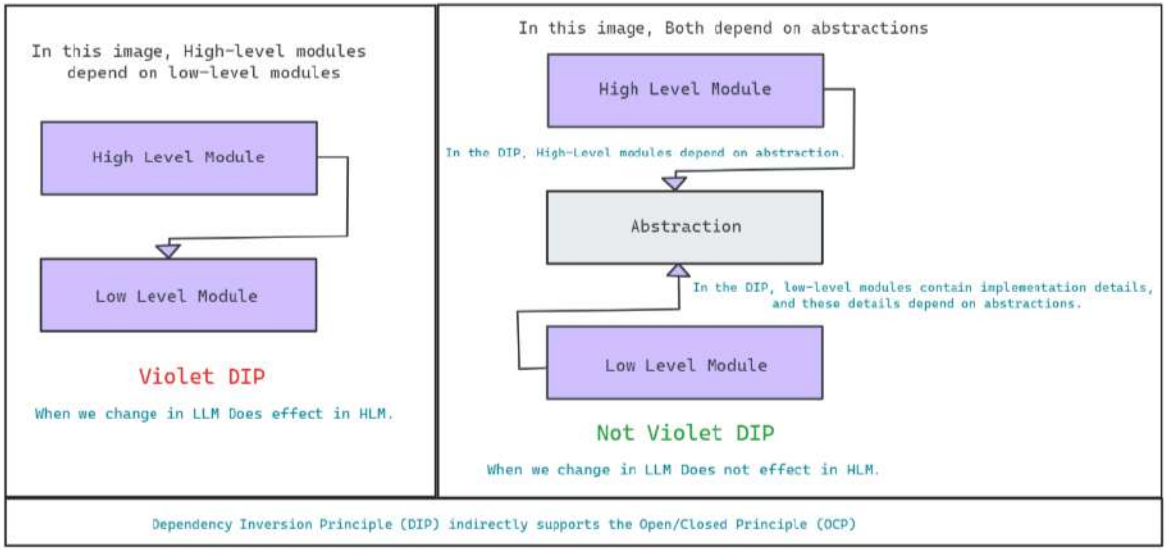
1. Never depend on everything **concrete(Actual Class)**, only depend on **Abstraction**.
2. High level module should not depend on **low level module**. They should depend on Abstraction.
3. Able to change an implementation easily without altering the high level code.
4. By adhering to **DIP**, you can create systems that are resilient to change, as modifications to concrete implementations do not affect **high-level modules**.

In One Statement

*The Dependency Inversion Principle suggests that **high-level modules** should not depend on **low-level modules**, but both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.*

Key Idea

1. ***High-level modules** should not depend on **low-level modules**; both should depend on abstractions.*
2. ***Abstractions** should not depend on **details**. Details should depend on abstractions.*



Real-Time Examples

- *Building a **LEGO tower** – the bricks **(high and low-level modules)** connect through smaller bricks **(abstractions)**.*

How can Interface Segregation Principle be applied?

Visit GitHub: @BCAPATHSHALA

Practical Coding Examples in Java #1
Practical Coding Examples in Java #2
Practical Coding Examples in Java #3
*Much more about **Dependency Inversion Principle***