DESIGN PATTERNS

1.SOLID PRINCIPLES

|  |
| --- |
| i. S-Single responsibility principle  The **Single Responsibility Principle (SRP)** means that a class should have **only one responsibility** and **one reason to change**. When a class takes on multiple responsibilities, it becomes more **complex**, harder to **maintain**, and less **reusable**.   **Better Maintainability** → Changes in one function won’t accidentally break another.   **Improved Reusability** → Each class can be reused independently in different contexts.   **Easier Debugging** → With focused classes, finding and fixing bugs becomes simpler.   **More Scalable Code** → The system remains flexible as requirements evolve.  So, if a class has **two unrelated responsibilities**, **the best approach** is to **split it into separate classes**, each handling a single responsibility. This keeps the code **modular** and **well-structured**.  Eg-printing and saving file into 2 differnet classes. |

|  |
| --- |
| ii. O-The **Open/Closed Principle (OCP)**  The **Open/Closed Principle (OCP)** states that **software entities should be open for extension but closed for modification**.   **Open for Extension** → You can add new functionality without altering existing code.   **Closed for Modification** → Instead of modifying existing code, extend it using **inheritance, abstraction, or composition**.   **Enhances Maintainability** → Reduces risk of breaking existing code when new features are introduced.  The **Open/Closed Principle (OCP)** encourages developers to **extend** functionality rather than **modify** existing code. By using **polymorphism**, we can create multiple subclasses that add new behaviors without altering the original class.  This approach keeps the main class **stable**, **reduces risk of breaking the code**, and makes the system more **scalable**. The beauty of OCP is that **new features** can be added  **effortlessly**—just by creating new subclasses instead of changing core logic.  Eg-discount if there are 2 discounts for normal and premium in main class used in if-else  If we need to add vip we have to modify so,rather we can extend and make a diff class for vip |
| iii. Liskov Substitution Principle (LSP)  LSP states that **subtypes must be substitutable for their base types** without altering the correctness of the program.  The **Liskov Substitution Principle (LSP)** states that **subclasses should be replaceable by their parent classes without altering the expected behavior** of the program.  **Key Points of LSP:**  ✅ Subtypes **must behave like the parent class** without introducing unexpected behavior. ✅ Derived classes **shouldn’t override functionality in a way that contradicts the parent class’s intended use**. ✅ If a subclass changes something, it **should not break** the system when substituted for the parent class. |

|  |
| --- |
| iv. Interface Segregation Principle (ISP)  **The Interface Segregation Principle (ISP) states that we should not create large, all-encompassing interfaces that force classes to implement methods they don’t need. Instead, we should break down interfaces into smaller, more specific ones, ensuring that each class only implements what it actually requires.**  **Reducing unnecessary dependencies**  **✅ Improving flexibility & modularity**  **✅ Making code easier to maintain & update**  If an interface has **too many methods**, some classes may be **forced to implement methods they don’t need**, leading to **unnecessary dependencies**.  Instead, breaking the interface into **smaller, more specific interfaces** allows classes to implement **only what is relevant to them**, making the system **more modular, maintainable, and flexible**.  that if there are many methods in an interfaces and a class needs only 1 and doesnt need other methods then its better to make different smaller interfaces so it becomes easier  example – interface - worker and eat ,ex-robot if extends interface,it doesn’t need eat it is forced to have so we can use create 2 differnet interfaces and extend whichever needed. |
| v. **Dependency Inversion Principle (DIP)**  DIP states that:   1. **High-level modules** should not depend on **low-level modules**. Both should depend on **abstractions**. 2. **Abstractions should not depend on details**. Instead, **details should depend on abstractions**.    **High-level modules should not directly depend on low-level modules.** Instead, both should depend on an **abstraction** (like an interface or an abstract class).   **Abstractions should not depend on details.** This means that **low-level implementations should depend on abstractions**, allowing flexibility and easier modifications.  By applying DIP, we make the system more **loosely coupled**, **scalable**, and **adaptable to changes** without breaking existing functionality.  that a high level module shouldnt be depend on low level module and also an abstartion is which used by low level and high level to depend  eg-msg and notification the email,sms are low level and notification is high level so they should be depend on abstract. |