**LLD AND HLD**

**1.LLD**

**SOLID PRINCIPLES:**

1. Single Class - Single Responsibility
2. Open for Extension, Close for Modification
3. Liskov Substitution
4. Interface segregation
5. Dependency Inversion

**1. Single Responsibility Principle (SRP)**

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

**Example**:



*In this example, Report class manages the content, while ReportPrinter is responsible for printing the report, adhering to SRP.*

**2. Open/Closed Principle (OCP)**

**Definition**: Software entities should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.

**Example**:

A screenshot of a computer program

Description automatically generated

*New shapes can be added (like Square) without modifying existing classes, fulfilling OCP.*

**3. Liskov Substitution Principle (LSP)**

**Definition**: Subtypes must be substitutable for their base types without affecting the correctness of the program.

**Example**:



*Instead, you can have a FlyingBird and a NonFlyingBird to adhere to LSP.*

**4. Interface Segregation Principle (ISP)**

**Definition**: Clients should not be forced to depend on interfaces they do not use. Instead of one large interface, multiple smaller, specific interfaces are better.

**Example**:

A screenshot of a computer program

Description automatically generated

*This way, a class can implement only what it needs, adhering to ISP.*

**5. Dependency Inversion Principle (DIP)**

**Definition**: High-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

**Example**:



*Here, UserService depends on the Database interface rather than a specific database implementation, adhering to DIP.*

**Summary**

* **SRP**: One class, one responsibility.
* **OCP**: Open for extension, closed for modification.
* **LSP**: Subtypes should be substitutable for their base types.
* **ISP**: No forced dependencies on unused methods.
* **DIP**: Depend on abstractions, not concretions.

By following these principles, you can create software that is more modular, easier to maintain, and adaptable to change.

**DESIGN PATTERNS**

1. *Creational Patterns*: Deal with object creation (e.g., Singleton, Factory Method).
2. *Structural Patterns*: Focus on how classes and objects are composed (e.g., Adapter, Decorator).
3. *Behavioral Patterns*: Concerned with object collaboration and responsibility (e.g., Strategy, Observer).

**Creational Patterns:**

1. Singleton:

 Ensures a class has only one instance and provides a global point of access to it.

 Example: Database connection class.



1. Factory Method:

 Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

 Example: Creating different shapes.



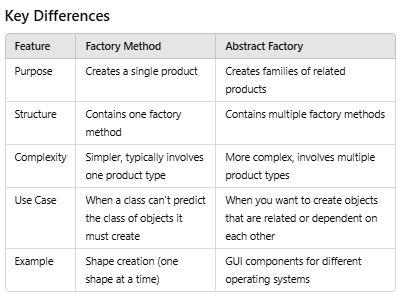
1. Abstract Factory Method

 Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

 Example: GUI components for different operating systems

A screenshot of a phone

Description automatically generated



4.

**Behavioural Patterns:**

1.Strategy Pattern:

**Key Components of the Strategy Pattern**

1. **Strategy Interface**: This interface defines a method that all concrete strategies will implement.
2. **Concrete Strategies**: These are specific implementations of the strategy interface, each encapsulating a different algorithm.
3. **Context**: This class uses a Strategy instance to call the algorithm defined by the Strategy interface. It maintains a reference to the Strategy interface

**Benefits of the Strategy Pattern**

* **Flexibility**: You can switch algorithms dynamically at runtime.
* **Encapsulation**: Each algorithm is encapsulated in its own class, promoting separation of concerns.
* **Reduced Conditional Logic**: It avoids the need for complex conditional statements (like if-else or switch) to select the algorithm.

**Example**

1. **Strategy Interface (SortStrategy)**: This interface defines a sort method that all concrete sorting strategies must implement.
2. **Concrete Strategies**:
   * BubbleSort and QuickSort implement the SortStrategy interface, each providing its own sorting algorithm.
3. **Context Class (SortContext)**: This class holds a reference to a SortStrategy object and uses it to perform the sorting. It allows setting the strategy dynamically.
4. **Client Code**: In the StrategyPatternExample class, we create a SortContext, set different sorting strategies, and sort arrays accordingly.

2. Observer Pattern:



**Key Components of the Observer Pattern**

1. **Subject**: This is the core object that holds the state and notifies observers about changes. It maintains a list of observers and provides methods for adding, removing, and notifying them.
2. **Observer**: This interface defines the method(s) that will be called when the subject changes its state. Concrete observers implement this interface.
3. **Concrete Subject**: A concrete implementation of the subject that maintains the state and notifies its observers of changes.
4. **Concrete Observer**: A specific implementation of the observer that reacts to state changes in the subject.

**Benefits of the Observer Pattern**

* **Loose Coupling**: The subject and observers are loosely coupled, meaning changes to one do not directly affect the other.
* **Dynamic Relationships**: Observers can be added or removed at runtime, allowing for flexible relationships.
* **Automatic Updates**: Observers are automatically notified when the subject changes state, reducing the need for manual checks.

