**SOLID PRINCIPLES :**

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code.

1. **Single Responsibility Principle :**

This principle states that “A class should have only one reason to change” which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

Ex :

the Employee class has a few employee class-specific behaviors like getDesignation & updateSalary.

Additionally, it also has another method named sendMail which deviates from the responsibility of the Employee class.

1. **Open/Closed Principle**

entity should be open for extension but closed for modification. A class can be extended via Inheritance, Interfaces, Composition whenever required instead of modifying the code of the class.

EX:

Consider an instance where we have a class that calculates the area of a square. Later, we get the requirement of calculating the area of a rectangle. Here, instead of modifying the original class, we can create one base class and this base class can be extended by the new class rectangle.

1. **Liscov Substitution Principle**

Derived or child classes must be substitutable for their base or parent classes“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

1. **Interface Segregation Principle**

Clients should not be forced to depend on interfaces they do not use.

we can use as many interfaces specific to the client’s requirements instead of creating only one general interface.

Ex:

abstract class FlyingBird{

abstract void fly();

}

abstract class NonFlyingBird{

abstract void doSomething();

}

class Eagle extends FlyingBird {

@Override

public void fly() { // some implementation }

}

class Ostrich extends NonFlyingBird {

@Override

public void doSomething() { // some implementation }

}

**Changing into** --------------------  
interface IAreaCalculator {

double calculateArea();

}

interface IVolumeCalculator {

double calculateVolume();

}

class Square implements IAreaCalculator {

@Override

public double calculateArea() { // calculate the area }

}

class Cube implements IAreaCalculator, IVolumeCalculator {

@Override

public double calculateArea() { // calculate the area }

@Override

public double calculateVolume() {// calculate the volume }

}

1. **Dependency Inversion**

 the high-level modules should not be dependent on the lower level modules or concrete implementations. Instead, they should be dependent on the abstractions

**Design Patterns in Java**

* Design patterns are reusable solutions to common software design problems.
* they provide ready-made and flexible solutions to common problems that software developers face.
* These solutions make the software better. Design patterns are focused on fixing problems in a specific type of software design, like how objects are organized and used.
* So that developers can fully focused on Core functionality of a application.

Why Categorize Design Patterns?

* **Identify appropriate patterns:** We can quickly narrow down the possible patterns based on the problem we are trying to solve.
* **Learn and understand patterns:** Grouping patterns by type helps us see patterns within patterns and understand their relationships.
* **Apply patterns effectively:** Knowing the type of pattern can guide us in how to use it and what benefits to expect.
* **Communicate and collaborate:** Using a common categorization system facilitates discussions and collaboration among developers.

By understanding these categories, we can more effectively use design patterns to create robust, maintainable, and scalable software.

1. **Creational Pattern** : These patterns deal with object creation. They aim to abstract the process of object creation, making the code more flexible and easier to maintain.
   * Singleton, Factory, Abstract Factory, Prototype, Builder
2. **Structural Patterns** : These patterns deal with how classes and objects are composed to form larger structures. They focus on relationships between objects.
   * Adapter, Bridge, Composite, Decorator, Façade, etc
3. **Behavioral Patterns :** These patterns deal with how objects interact with each other and their responsibilities. They focus on the communication and collaboration between objects.
   * Template method, Strategy, Iterator, State, Command, Interpreter, etc

**What is a Singleton Pattern?**

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

 **Private Constructor:** The constructor is made private to prevent direct instantiation from outside the class.

 **Static Instance Variable:** A static variable instance is declared to hold the single instance of the class.

 **Static getInstance() Method:** This method provides a public point of access to the instance. It checks if the instance exists and creates it if necessary using double-checked locking to ensure thread safety.

EX: 1)log file in spring application single instance is created for all logs

2)data source for database connection

3) properties file for configuration

public class Employee {

private static Employee instance;

private Employee() {

// Private constructor to prevent external instantiation

}

public static Employee getInstance() {

if (instance == null) {

synchronized (Employee.class) {

if (instance == null) {

instance = new Employee();

}

}

}

return instance;

}

// Employee methods and attributes here

public void doWork() {

// ...

}

}

2. Factory:

* Used when we have multiple sub-classes of a super class & based on input we want to return one class instance
* It provides abstraction between implementation & client classes

Super class can be interface or abstract class or basic class

Factory class has a static method which returns the instance of sub class based on input

Ex: **abstract class**Vehicle {  
  **public abstract int**getWheel();  
    
  **public**String toString() {  
    **return**"Wheel: " + **this**.getWheel();  
  }  
}  
  
**class**Car **extends**Vehicle {  
  **int**wheel;  
    
  Car(**int**wheel) {  
    **this**.wheel = wheel;  
  }  
  
  @Override  
  **public int**getWheel() {  
    **return this**.wheel;  
  }  
}  
  
**class**Bike **extends**Vehicle {  
  **int**wheel;  
    
  Bike(**int**wheel) {  
    **this**.wheel = wheel;  
  }  
    
  @Override  
  **public int**getWheel() {  
    **return this**.wheel;  
  }  
}  
  
**class**VehicleFactory {  
  **public static**Vehicle getInstance(String type, **int**wheel) {  
    **if**(type == "car") {  
      **return new**Car(wheel);  
    } **else if**(type == "bike") {  
      **return new**Bike(wheel);  
    }  
      
    **return null**;  
  }  
}  
  
**public class**FactoryPatternExample {  
  
  **public static void**main(String[] args) {  
    Vehicle car = VehicleFactory.getInstance("car", 4);  
    System.out.println(car);  
      
    Vehicle bike = VehicleFactory.getInstance("bike", 2);  
    System.out.println(bike);  
  }  
  
}

3. Builder Pattern :

* Used when we have too many arguments to send in constructor and its hard to maintain the order

Generally we send optional parameters as null

* We create a static nested class which contains all arguments of outer class
* As per naming convertion if class name is vehicle, builder class should be vehicleBuilder
* Builder class have a public constructor with all required parameters
* Builder class should have methods for optional parameters
* A build() method that will return the final object