**Principles of class design**

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.

* **A class should have single responsibility.**
* **A responsibility should be defined as a reason for change.**
* **A class should have one and only one reason to change.**

**Example:**

In this example single class has three responsibilities due to which it will get more request changes and affect other responsibilities also.

A picture containing timeline

Description automatically generated

In the next image we can see that for all 3 responsibilities we make three different classes and this is srp.

Graphical user interface, diagram

Description automatically generated with medium confidence

1. **2. Open/Closed Principle:**This principle states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behavior, without modifying it.

**Example:**

**If the below example if we add some other functionalities in same class then it will be a bad design.**

Graphical user interface, application

Description automatically generated

**->** Ocp can be achieved by using virtual keyword in parent class function then in child class we can made same function using virtual keyword

Graphical user interface, text, application

Description automatically generated

1. **Liskov’s Substitution Principle:**The principle was introduced by Barbara Liskov in 1987 and according to this 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.

**Example:**

In the below example if we want to excess GetLeavesDetail function using these derived classes instances then they will throw an error because it is throwing an error in the TraineeEmployee class.

Graphical user interface, text, application

Description automatically generated

**Solution:** We can make 2 interfaces and the class which needs both can inhertit both interfaces and the class which need only one can inherit only one.Graphical user interface, text, application

Description automatically generated

1. **Interface Segregation Principle:**This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “do not force any client to implement an interface which is irrelevant to them“.

**Example:**

Graphical user interface, text, application

Description automatically generated

**Solution :**

We can create interface for two different classes and if the class needs both functions than both interfaces can be used and icp achieved.Graphical user interface, text, application, email

Description automatically generated

**Cohesion and Coupling [context of modules and context of classes]**

1. **Cohesion:**

Cohesion in Java is the Object-Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose. In object-oriented design, cohesion refers to how a single class is designed.

**Example:**Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don’t have much in common. To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply’s method to get the result and display it. This way to develop a high cohesive solution.

|  |
| --- |
| // Java program to illustrate  // high cohesive behavior    **class** Multiply {    **int** a = 5;  **int** b = 5;    **public** **int** mul(**int** a, **int** b)      {  **this**.a = a;  **this**.b = b;  **return** a \* b;      }  }    **class** Display {  **public** **static** **void** main(String[] args)      {          Multiply m = **new** Multiply();          System.out.println(m.mul(5, 5));      }  } |

1. **Coupling :**

In object oriented design, Coupling refers to the degree of direct knowledge that one element has of another. In other words, how often do changes in class A force related changes in class B.  
**There are two types of coupling:**

* 1. **Tight coupling :**In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.  
     **Example :**If you want to change the skin, you would also have to change the design of your body as well because the two are joined together – they are tightly coupled. The best example of tight coupling is RMI(Remote Method Invocation).

|  |
| --- |
| // Java program to illustrate  // tight coupling concept  **class** Subject {      Topic t = **new** Topic();  **public** **void** startReading()      {          t.understand();      }  }  **class** Topic {  **public** **void** understand()      {          System.out.println("Tight coupling concept");      }  } |

**Explanation:** In the above program the Subject class is dependents on Topic class. In the above program Subject class is tightly coupled with Topic class it means if any change in the Topic class requires Subject class to change. For example, if Topic class understand() method change to gotit() method then you have to change the startReading() method will call gotit() method instead of calling understand() method.

* 1. **Loose coupling :**In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled. In order to over come from the problems of tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling.  
     **Example :**If you change your shirt, then you are not forced to change your body – when you can do that, then you have loose coupling. When you can’t do that, then you have tight coupling. The examples of Loose coupling are Interface, JMS.

|  |
| --- |
| // Java program to illustrate  // loose coupling concept  **public** **interface** Topic  {  **void** understand();  }  **class** Topic1 **implements** Topic {  **public** **void** understand()      {          System.out.println("Got it");      }  } **class** Topic2 **implements** Topic {  **public** **void** understand()      {          System.out.println("understand");      }  } **public** **class** Subject {  **public** **static** **void** main(String[] args)      {          Topic t = **new** Topic1();          t.understand();      }  } |

**Explanation :** In the above example, Topic1 and Topic2 objects are loosely coupled. It means Topic is an interface and we can inject any of the implemented classes at run time and we can provide service to the end user.

**Relationship between classes - Association, composition, Aggregation**

Association is a relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



**Example:**

* Java

|  |
| --- |
| // Java Program to illustrate the  // Concept of Association    // Importing required classes  **import** java.io.\*;    // Class 1  // Bank class  **class** Bank {        // Attributes of bank  **private** String name;    **private** Set<Employee> employees;      // Constructor of this class      Bank(String name)      {          // this keyword refers to current instance itself  **this**.name = name;      }        // Method of Bank class  **public** String getBankName()      {          // Returning name of bank  **return** **this**.name;      }    **public** setEmployees(Set<Employee> employees){  **this**.employees = employees;      }  **public** getEmployees(Set<Employee> employees){  **return** **this**.employees;      }  }    // Class 2  // Employee class  **class** Employee {      // Attributes of employee  **private** String name;      // Employee name      Employee(String name)      {          // This keyword refers to current instance itself  **this**.name = name;      }        // Method of Employee class  **public** String getEmployeeName()      {          // returning the name of employee  **return** **this**.name;      }  }    // Class 3  // Association between both the  // classes in main method  **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)      {            // Creating objects of bank and Employee class          Bank bank = **new** Bank("ICICI");          Employee emp = **new** Employee("Ridhi");            Set<Employee> employees = **new** HashSet<>();            employees.add(emp);              bank.setEmployees(employees);            System.out.println(bank.getEmployees()+"are belongs to bank"+bank.getBankName());      }  } |

**Output:**

Ridhi is employee of ICICI

**Output Explanation:**In the above example, two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

Composition

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.y

**Example**

class Engine {

// fields for the engine

int horsepower;

String type;

}

class Car {

// fields for the car

String make;

String model;

// car has an engine

Engine engine;

// constructor for the Car class

public Car(String make, String model, int horsepower, String type) {

this.make = make;

this.model = model;

this.engine = new Engine();

this.engine.horsepower = horsepower;

this.engine.type = type;

}

// method to start the car

public void start() {

System.out.println("Starting car...");

}

// method to stop the car

public void stop() {

System.out.println("Stopping car...");

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car("Tesla", "Model S", 450, "electric");

// start the car

car.start();

// print out the make, model, horsepower, and type of the car's engine

System.out.println(car.make);

System.out.println(car.model);

System.out.println(car.engine.horsepower);

System.out.println(car.engine.type);

// stop the car

car.stop();

}

}

Output is:

Starting car...

Tesla

Model S

450

electric

Stopping car...

**Expalnation :** The **Car** class has complete ownership over the **Engine** object, which means that the **Engine** object cannot be used outside the **Car** class. This is why the **Engine** object is created inside the constructor of the **Car** class and its fields are not accessible directly from the **main** method.

**Aggregation**

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, a Team object and a Player object. The team contains multiple players but a player can exist without a team.

Example

class Address {

// fields for the address

String street;

String city;

String state;

String country;

int zip;

}

class Employee {

// name of the employee

String name;

// employee has an address

Address address;

// Employee has a non-static nested class Address

}

public class Main {

public static void main(String[] args) {

Employee employee = new Employee();

employee.name = "John Doe";

// create a new address for the employee

employee.address = new Address();

employee.address.street = "123 Main St.";

employee.address.city = "New York";

employee.address.state = "NY";

employee.address.country = "USA";

employee.address.zip = 10001;

// print out the employee's name and address

System.out.println(employee.name);

System.out.println(employee.address.street);

System.out.println(employee.address.city);

System.out.println(employee.address.state);

System.out.println(employee.address.country);

System.out.println(employee.address.zip);

}

}

**Output:**

John Doe

123 Main St.

New York

NY

USA

10001

**Explanation:**

In this example, the **Employee** class has an instance variable **address** of type **Address**, which represents the aggregation relationship between the two classes. The **Address** class is known as the aggregated class and the **Employee** class is the aggregating class.

The difference between aggregation and composition is that in aggregation, the aggregated class (in this case, **Address**) can exist independently of the aggregating class (in this case, **Employee**). In other words, an **Address** object can be created and used without an **Employee** object being present.

**Factory Design Pattern**

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Step 1

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

Step 3

Create a Factory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

*FactoryPatternDemo.java*

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of square

shape3.draw();

}

}

Step 5

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

**Unit testing - Check coverage of unit testing**

Every Software Developer follows [Software Development Life Cycle (SDLC)](https://www.geeksforgeeks.org/software-development-life-cycle-sdlc/) for the development of any software application. In which testing is one of the important phase which is performed to check whether the developed software application is fulfilling the requirements or not. Different [types of software testing](https://www.geeksforgeeks.org/types-software-testing/) are there which are performed based on various metrics/testing parameters.

**Code Coverage :**  
Code coverage is a software testing metric or also termed as a Code Coverage Testing which helps in determining how much code of the source is tested which helps in accessing quality of test suite and analyzing how comprehensively a software is verified. Actually in simple code coverage refers to the degree of which the source code of the software code has been tested. This Code Coverage is considered as one of the form of [white box testing.](https://www.geeksforgeeks.org/software-engineering-white-box-testing/)

As we know at last of the development each client wants a quality software product as well as the developer team is also responsible for delivering a quality software product to the customer/client. Where this quality refers to the product’s performance, functionalities, behavior, correctness, reliability, effectiveness, security, and maintainability. Where Code Coverage metric helps in determining the performance and quality aspects of any software.

The formula to calculate code coverage is

*Code Coverage = (Number of lines of code executed)/(Total Number of lines of code in a system component) \* 100*

**Code Coverage Criteria :**  
To perform code coverage analysis various criteria are taken into consideration. These are the major methods/criteria which are considered.

**1. Statement Coverage/Block coverage :**  
The number of statements that have been successfully executed in the program source code.

*Statement Coverage**= (Number of statements executed)/(Total Number of statements)\*100.*

**2. Decision Coverage/Branch Coverage :**  
The number of decision control structures that have been successfully executed in the program source code.

*Decision Coverage = (Number of decision/branch outcomes exercised)/(Total number of decision outcomes in the source code)\*100.*

**3. Function coverage :**  
The number of functions that are called and executed at least once in the source code.

*Function Coverage = (Number of functions called)/(Total number of function)\*100.*

**4. Condition Coverage/Expression Coverage :**  
The number of Boolean condition/expression statements executed in the conditional statement.

*Condition Coverage =(Number of executed operands)/(Total Number of Operands)\*100.*

**Tools For Code Coverage :**  
Below are the few important code coverage tools

* Cobertura
* Clover
* Gretel
* Kalistick
* JaCoCo
* JTest
* OpenCover
* Emma
* GCT

**Advantages of Using Code Coverage :**

* It helps in determining the performance and quality aspects of any software.
* It helps in evaluating quantitative measure of code coverage.
* It helps in easy maintenance of code base.
* It helps in accessing quality of test suite and analyzing how comprehensively a software is verified.
* It helps in exposure of bad, dead, and unused code.
* It helps in creating extra test cases to increase coverage.
* It helps in developing the software product faster by increasing its productivity and efficiency.
* It helps in measuring the efficiency of test implementation.
* It helps in finding new test cases which are uncovered.