**Design patterns** represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**.

# **Core Java Design Patterns**

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

## 1.Creational Design Pattern : These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.

1. Factory Pattern
2. Abstract Factory Pattern
3. **Singleton Pattern**
4. Prototype Pattern
5. **Builder Pattern.**

## 2. Structural Design Pattern: These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

1. **Adapter Pattern**
2. Bridge Pattern
3. **Composite Pattern**
4. **Decorator Pattern**
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

## 3. Behavioral Design Pattern: These design patterns are specifically concerned with communication between objects.

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. **Observer Pattern**
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. **Visitor Pattern**

**Singleton pattern** is one of the simplest 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.

This pattern involves a single class which is responsible to create an object while making sure that **only single object gets created**. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



Step 1

Create a Singleton Class.

*SingleObject.java*

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

Step 2

Get the only object from the singleton class.

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object1 = SingleObject.getInstance();

//show the message

object1.showMessage();

SingleObject object2 = SingleObject.getInstance();

Object2.showMessage();

}

}

Step 3

Verify the output.

Hello World!

## What is adapter design pattern?

Adapter design pattern in java is a [structural design pattern](https://stacktraceguru.com/software-design-patterns). It provides solution for helping incompatible things to communicate with each other. It works as an inter-mediator who takes output from one client and gives it to other after converting in the expected format. Adapter pattern is also known as wrapper.

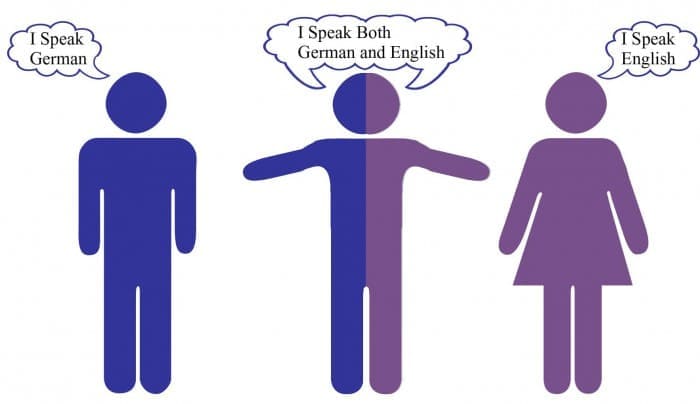
Adapter design pattern structure

Flow diagram of Adapter design pattern

The figure above shows the basic structure and flow of the adapter design pattern. We can see, there are 2 processes Process1 producing OutputA and Process2 that requires input from Process1. But in some cases, the output produced by Process1 is not of the same form as that required by Process2. Hence, An adapter is introduced between both the processes. What is the purpose of this adapter? It is nothing but a piece of code or a process that converts OutputA into a format which Process2 can use i.e OutputB.

Real life example of adapter design pattern

There are so many examples where we use adapter patterns. One of the example we can consider of language translator.



As shown in image there are two people one can speak English and other speaks German. Therefore, there is no way they can communicate. Then they find another person who speaks both the language who helps them communicate.

Adapter design pattern example in software

Let’s assume an example of a student who becomes an employee after candidate selection process.

We have two model objects Student.java and Employee.java. Student model is for storing the student data. Employee model stores the employee data.

public class Student {

int rollNumber;

String name;

Date dob;

.. getter and setter ...

}

public class Employee{

int employeeId;

String name;

Date dateOfBirth;

public Employee(int employeeId, String name, Date dateOfBirth) {

this.employeeId = employeeId;

this.name = name;

this.dateOfBirth = dateOfBirth;

}

.. getter and setter ...

}

We have a EmployeeService.java class. This class creates as well as stores employee data.

public class EmployeeService{

public void saveEmployee(Employee e){

.. code to create employee record

}

}

Now let’s create the code for interview selection process. The input will be student object. If interview is successful then we need to create employee record.

public class PlacementService{

EmployeeService employeeService = new EmployeeService();

public void process(Student student){

// conduct interview

// if selected in interview

employeeService.saveEmployee(EmployeeAdaptor.getEmployee(student));

}

}

public class EmployeeAdapter{

public static Employee getEmployee(Student s){

return new Employee(s.getRollNumber(),s.name,s.getDob());

}

}

**Builder Pattern in Java – with examples**

Builder design pattern is one of the most important and simple design pattern in java.

We use it in real life also. For instance, consider assembling a car or making a burger. It is similar to making an object of a class in Object Oriented Programming for example navigation menu of any application.

Where to use Builder pattern?

What to solve?

The Builder design pattern solves problems like:

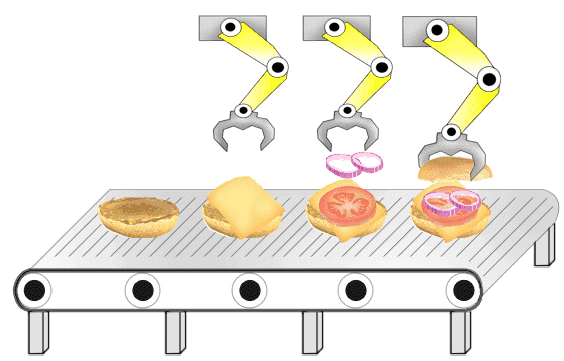
* How can a class (the same construction process) create different representations of a complex object?
* How can a class that includes creating a complex object be simplified?

How to solve?

The Builder design pattern describes how to solve such problems:

A class delegates object creation to a Builder object instead of creating the objects directly.

Encapsulate creating and assembling the parts of a complex object in a separate Builder object.

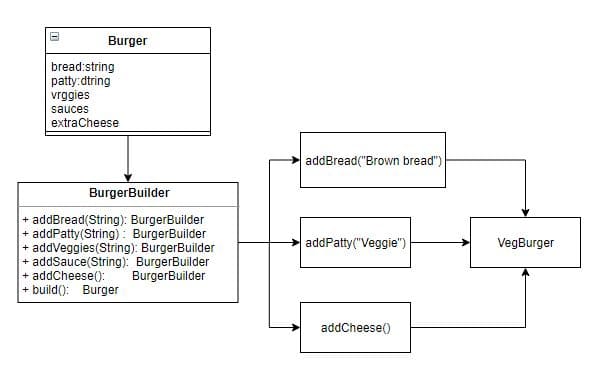


How does Builder pattern work?

Let us divide the code in 3 parts to understand how this pattern works. Then we will see the significance of each part.

1. **Product :** This part of the code is the actual object that we are trying to build. For example, in our case it would be ‘the burger’.
2. **Builder :** It will contain the general methods needed to build the product.
3. **Executor :**This is the important part which actually calls the builder methods to create the ‘Product’. This in our case would be the ‘Employee’ who prepares the burger for us. He/She will invoke appropriate method from the ‘BurgerBuilder’

Implementation of Builder pattern



Let us now see the implementation of this pattern in Java. We will see each part one by one starting with the “Product”, which in our case is the “Burger”. This is just a normal entity of which we need to create different instances as per our requirement.

package main.java.creational.builder;

public class Burger {

private String bread;

private String patty;

private String veggies;

private String sauces;

private Boolean withExtraCheese = Boolean.FALSE;

public String getBread() { return bread; }

public void setBread(String bread) { this.bread = bread; }

<......... Getters and setters for these properties>

@Override

public String toString() {

return "Burger [bread=" + bread + ", patty=" + patty + ",

veggies=" + veggies + ", sauces=" + sauces + ",

withExtraCheese=" + withExtraCheese + "]";

}

}

The next part of our pattern is the “Builder” itself. As the name suggests, this is the class contains the methods we need to build the product.

package main.java.creational.builder;

public class BurgerBuilder {

private Burger burger;

public BurgerBuilder () { this.burger = new Burger(); }

public void addBread(String bread) { burger.setBread(bread); }

public void addPatty(String patty ) { burger.setPatty(patty); }

public void addVeggies(String veggies ) { burger.setVeggies(veggies ); }

public void addSauce(String sauce) { burger.setSauces(sauce); }

public void addCheeze() { burger.setWithExtraCheese(Boolean.TRUE); }

public Burger build() { return this.burger; }

}

Now, we come to the last but important part of our code. This is called as the “Executor”. Executor calls the builder methods to create the product.  
Thus we will have an “Employee” to create the veg burger of us in our example. Let us take a look at how our Employee looks like! 😛

package main.java.creational.builder;

public class Employee {

public static void main(String[] args) {

Burger vegBurger = new BurgerBuilder().addBread("Brown Bread").addPatty("Veggie").addVeggies("Pickles").addSauce("All").addCheeze().build();

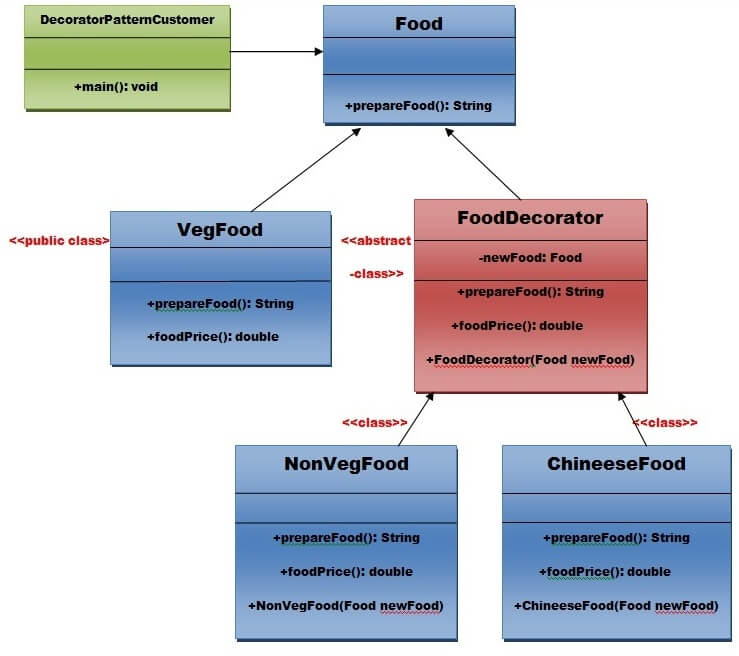
}

}

**Decorator pattern** allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

#### **UML for Decorator Pattern:**



Step 1:**Create a Food interface.**

1. **public** **interface** Food {
2. **public** String prepareFood();
3. **public** **double** foodPrice();
4. }// End of the Food interface.

Step 2: Create a **VegFood** class that will implements the **Food** interface and override its all methods.

*File: VegFood.java*

1. **public** **class** VegFood **implements** Food {
2. **public** String prepareFood(){
3. **return** "Veg Food";
4. }
6. **public** **double** foodPrice(){
7. **return** 50.0;
8. }
9. }

Step 3:Create a FoodDecorator abstract class that will implements the Food interface and override it's all methods and it has the ability to decorate some more foods.

*File: FoodDecorator.java*

1. **public** **abstract** **class** FoodDecorator **implements** Food{
2. **private** Food newFood;
3. **public** FoodDecorator(Food newFood)  {
4. **this**.newFood=newFood;
5. }
6. @Override
7. **public** String prepareFood(){
8. **return** newFood.prepareFood();
9. }
10. **public** **double** foodPrice(){
11. **return** newFood.foodPrice();
12. }
13. }

Step 4:Create a **NonVegFood concrete** class that will extend the **FoodDecorator** class and override it's all methods.

*File: NonVegFood.java*

1. **public** **class** NonVegFood **extends** FoodDecorator{
2. **public** NonVegFood(Food newFood) {
3. **super**(newFood);
4. }
5. **public** String prepareFood(){
6. **return** **super**.prepareFood() +" With Roasted Chiken and Chiken Curry  ";
7. }
8. **public** **double** foodPrice()   {
9. **return** **super**.foodPrice()+150.0;
10. }
11. }

Step 5:Create a **ChineeseFood** concrete class that will extend the **FoodDecorator** class and override it's all methods.

*File: ChineeseFood.java*

1. **public** **class** ChineeseFood **extends** FoodDecorator{
2. **public** ChineeseFood(Food newFood)    {
3. **super**(newFood);
4. }
5. **public** String prepareFood(){
6. **return** **super**.prepareFood() +" With Fried Rice and Manchurian  ";
7. }
8. **public** **double** foodPrice()   {
9. **return** **super**.foodPrice()+65.0;
10. }
11. }

Step 6:Create a **DecoratorPatternCustomer** class that will use Food interface to use which type of food customer wants means (Decorates).

*File: DecoratorPatternCustomer.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **public** **class** DecoratorPatternCustomer {
5. **private** **static** **int**  choice;
6. **public** **static** **void** main(String args[]) **throws** NumberFormatException, IOException    {
7. **do**{
8. System.out.print("========= Food Menu ============ \n");
9. System.out.print("            1. Vegetarian Food.   \n");
10. System.out.print("            2. Non-Vegetarian Food.\n");
11. System.out.print("            3. Chineese Food.         \n");
12. System.out.print("            4. Exit                        \n");
13. System.out.print("Enter your choice: ");
14. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
15. choice=Integer.parseInt(br.readLine());
16. **switch** (choice) {
17. **case** 1:{
18. VegFood vf=**new** VegFood();
19. System.out.println(vf.prepareFood());
20. System.out.println( vf.foodPrice());
21. }
22. **break**;
24. **case** 2:{
25. Food f1=**new** NonVegFood((Food) **new** VegFood());
26. System.out.println(f1.prepareFood());
27. System.out.println( f1.foodPrice());
28. }
29. **break**;
30. **case** 3:{
31. Food f2=**new** ChineeseFood((Food) **new** VegFood());
32. System.out.println(f2.prepareFood());
33. System.out.println( f2.foodPrice());
34. }
35. **break**;
37. **default**:{
38. System.out.println("Other than these no food available");
39. }
40. **return**;
41. }//end of switch
43. }**while**(choice!=4);
44. }
45. }

[download this Decorator Pattern Example](https://www.javatpoint.com/designpattern/designpatternexample/decoratorpattern.zip)

#### **Output**

1. ========= Food Menu ============
2. 1. Vegetarian Food.
3. 2. Non-Vegetarian Food.
4. 3. Chineese Food.
5. 4. Exit
6. Enter your choice: 1
7. Veg Food
8. 50.0
9. ========= Food Menu ============
10. 1. Vegetarian Food.
11. 2. Non-Vegetarian Food.
12. 3. Chineese Food.
13. 4. Exit
14. Enter your choice: 2
15. Veg Food With Roasted Chiken and Chiken Curry
16. 200.0
17. ========= Food Menu ============
18. 1. Vegetarian Food.
19. 2. Non-Vegetarian Food.
20. 3. Chineese Food.
21. 4. Exit
22. Enter your choice: 3
23. Veg Food With Fried Rice and Manchurian
24. 115.0
25. ========= Food Menu ============
26. 1. Vegetarian Food.
27. 2. Non-Vegetarian Food.
28. 3. Chineese Food.
29. 4. Exit
30. Enter your choice: 4
31. Other than these no food available

**Observer pattern** is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.

*ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



Step 1

Create Subject class.

*Subject.java*

import java.util.ArrayList;

import java.util.List;

public class Subject {

private List<Observer> observers = new ArrayList<Observer>();

private int state;

public int getState() {

return state;

}

public void setState(int state) {

this.state = state;

notifyAllObservers();

}

public void attach(Observer observer){

observers.add(observer);

}

public void notifyAllObservers(){

for (Observer observer : observers) {

observer.update();

}

}

}

Step 2

Create Observer class.

*Observer.java*

public abstract class Observer {

protected Subject subject;

public abstract void update();

}

Step 3

Create concrete observer classes

*BinaryObserver.java*

public class BinaryObserver extends Observer{

public BinaryObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );

}

}

*OctalObserver.java*

public class OctalObserver extends Observer{

public OctalObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );

}

}

*HexaObserver.java*

public class HexaObserver extends Observer{

public HexaObserver(Subject subject){

this.subject = subject;

this.subject.attach(this);

}

@Override

public void update() {

System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );

}

}

Step 4

Use *Subject* and concrete observer objects.

*ObserverPatternDemo.java*

public class ObserverPatternDemo {

public static void main(String[] args) {

Subject subject = new Subject();

new HexaObserver(subject);

new OctalObserver(subject);

new BinaryObserver(subject);

System.out.println("First state change: 15");

subject.setState(15);

System.out.println("Second state change: 10");

subject.setState(10);

}

}

Step 5

Verify the output.

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

Octal String: 12

Binary String: 1010

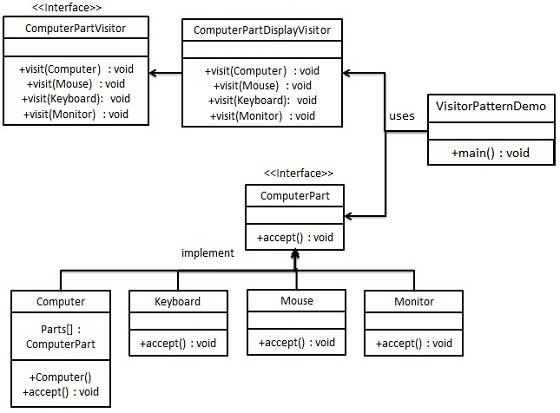
**In Visitor pattern,** we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

“**Visitor** is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.”

Implementation

We are going to create a *ComputerPart* interface defining accept opearation. *Keyboard*, *Mouse*, *Monitor* and *Computer* are concrete classes implementing *ComputerPart* interface. We will define another interface *ComputerPartVisitor* which will define a visitor class operations. *Computer* uses concrete visitor to do corresponding action.

*VisitorPatternDemo*, our demo class, will use *Computer* and *ComputerPartVisitor* classes to demonstrate use of visitor pattern.



Step 1

Define an interface to represent element.

*ComputerPart.java*

public interface ComputerPart {

public void accept(ComputerPartVisitor computerPartVisitor);

}

Step 2

Create concrete classes extending the above class.

*Keyboard.java*

public class Keyboard implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

*Monitor.java*

public class Monitor implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

*Mouse.java*

public class Mouse implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

*Computer.java*

public class Computer implements ComputerPart {

ComputerPart[] parts;

public Computer(){

parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};

}

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

for (int i = 0; i < parts.length; i++) {

parts[i].accept(computerPartVisitor);

}

computerPartVisitor.visit(this);

}

}

Step 3

Define an interface to represent visitor.

*ComputerPartVisitor.java*

public interface ComputerPartVisitor {

public void visit(Computer computer);

public void visit(Mouse mouse);

public void visit(Keyboard keyboard);

public void visit(Monitor monitor);

}

Step 4

Create concrete visitor implementing the above class.

*ComputerPartDisplayVisitor.java*

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

@Override

public void visit(Computer computer) {

System.out.println("Displaying Computer.");

}

@Override

public void visit(Mouse mouse) {

System.out.println("Displaying Mouse.");

}

@Override

public void visit(Keyboard keyboard) {

System.out.println("Displaying Keyboard.");

}

@Override

public void visit(Monitor monitor) {

System.out.println("Displaying Monitor.");

}

}

Step 5

Use the *ComputerPartDisplayVisitor* to display parts of *Computer*.

*VisitorPatternDemo.java*

public class VisitorPatternDemo {

public static void main(String[] args) {

ComputerPart computer = new Computer();

computer.accept(new ComputerPartDisplayVisitor());

}

}

Step 6

Verify the output.

Displaying Mouse.

Displaying Keyboard.

Displaying Monitor.

Displaying Computer.

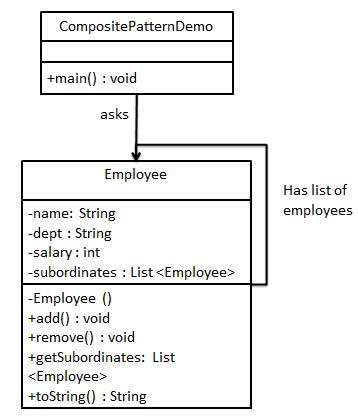
**Composite pattern** is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

We are demonstrating use of composite pattern via following example in which we will show employees hierarchy of an organization.

Implementation

We have a class *Employee* which acts as composite pattern actor class. *CompositePatternDemo*, our demo class will use *Employee* class to add department level hierarchy and print all employees.



Step 1

Create *Employee* class having list of *Employee* objects.

*Employee.java*

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private String dept;

private int salary;

private List<Employee> subordinates;

// constructor

public Employee(String name,String dept, int sal) {

this.name = name;

this.dept = dept;

this.salary = sal;

subordinates = new ArrayList<Employee>();

}

public void add(Employee e) {

subordinates.add(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

public List<Employee> getSubordinates(){

return subordinates;

}

public String toString(){

return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");

}

}

Step 2

Use the *Employee* class to create and print employee hierarchy.

*CompositePatternDemo.java*

public class CompositePatternDemo {

public static void main(String[] args) {

Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);

Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);

CEO.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

//print all employees of the organization

System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {

System.out.println(headEmployee);

for (Employee employee : headEmployee.getSubordinates()) {

System.out.println(employee);

}

}

}

}

Step 3

Verify the output.

Employee :[ Name : John, dept : CEO, salary :30000 ]

Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]

Employee :[ Name : Richard, dept : Sales, salary :10000 ]

Employee :[ Name : Rob, dept : Sales, salary :10000 ]

Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]

Employee :[ Name : Laura, dept : Marketing, salary :10000 ]

Employee :[ Name : Bob, dept : Marketing, salary :10000 ]

# **SOLID Principle in Programming: Understand With Real Life Examples**

In software development, **Object-Oriented Design** plays a crucial role when it comes to writing flexible, scalable, maintainable, and reusable code. There are so many benefits of using OOD but every developer should also have the knowledge of the SOLID principle for good object-oriented design in programming. The SOLID principle was introduced by ***Robert C. Martin***, also known as Uncle Bob and it is a coding standard in programming. This principle is an acronym of the five principles which is given below…

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle
3. Liskov’s Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

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. Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes. Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

1. **Single Responsibility Principle:** every class should have a single responsibility or single job or single purpose.

Take the example of developing software. The task is divided into different members doing different things as front-end designers do design, the tester does testing and backend developer takes care of backend development part then we can say that everyone has a single job or responsibility.  
Most of the time it happens that when programmers have to add features or new behavior they implement everything into the existing class which is completely wrong. It makes their code lengthy, complex and consumes time when later something needs to be modified. Use layers in your application and break God classes into smaller classes or modules.

**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.  
Suppose developer A needs to release an update for a library or framework and developer B wants some modification or add some feature on that then developer B is allowed to extend the existing class created by developer A but developer B is not supposed to modify the class directly. Using this principle separates the existing code from the modified code so it provides better stability, maintainability and minimizes changes as in your code.

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.  
You can understand it in a way that a farmer’s son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father but if he wants to become a cricketer then definitely the son can’t replace his father even though they both belong to the same family hierarchy.

One of the classic examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class. In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle. Check the link [Liskov Substitution Principle](https://www.youtube.com/watch?v=Jecou7B3nhc" \t "_blank) for better understanding.

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“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.  
   Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets. In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don’t eat in your food. Here the menu should be different for different types of customers. The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

**5. Dependency Inversion Principle:**Before we discuss this topic keep in mind that Dependency Inversion and [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection) both are different concepts. Most of the people get confused about it and consider both are the same. Now two key points are here to keep in mind about this principle

* High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.

The above lines simply state that if a high module or class will be dependent more on low-level modules or class then your code would have tight coupling and if you will try to make a change in one class it can break another class which is risky at the production level. So always try to make classes loosely coupled as much as you can and you can achieve this through abstraction. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.  
You can consider the real-life example of a TV remote battery. Your remote needs a battery but it’s not dependent on the battery brand. You can use any XYZ brand that you want and it will work. So we can say that the TV remote is loosely coupled with the brand name. Dependency Inversion makes your code more reusable.