Design pattern and SOLID

In Java design pattern are basically divided in 3 basic category:

* Creational design pattern
* Structural design pattern.
* Behavioral design pattern.

Creational design pattern:

Following design pattern comes under the creational design pattern :

**1.) Singleton design pattern.**

**2.) Factory design pattern**

**3.) Abstract factory design pattern.**

**4.) Prototype design pattern.**

Structural design pattern:

Following design pattern comes under the structural design pattern.

**1.) Decorator**

**2.) Composite**

**3.) Adapter**

**4.) Bridge**

**5.) Flyweight**

**6.) Proxy**

**7.) Façade**

Behavioral design pattern:

**1.) Chain of responsibility**

**2.) Command**

**3.) Template method**

**4.) Strategy**

**5.) State**

**6.) Iterator**

**7.) Mediator**

**8.) Observer**

**9.) Visitor**

**10.) Memento**

**11.) Interpreter.**

# Singleton design Pattern**:**

singleton design pattern is basically used when we needed single object in the application.

In other words Singleton design patterns says define a class that has only one instance and provides a Global access point to access the same.

Only single instance should be created and single object can be used by all other classes.

There are 2 forms of singleton design pattern.

Early installation : creation of instance at load time.

Lazy installation : creation of the instance when required.

Advantages :

Saves memory as only single instance is created and used again and again instead of creating the new instance.

Singleton thread application is mostly used in multi- threaded application and database application. It is mostly used for caching, logging purpose.

### How to create Singleton design pattern?

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

### **Understanding early Instantiation of Singleton Pattern**

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of class loading.

Let's see the example of singleton design pattern using early instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj=**new** A();//Early, instance will be created at load time
3. **private** A(){}
5. **public** **static** A getA(){
6. **return** obj;
7. }
9. **public** **void** doSomething(){
10. //write your code
11. }
12. }

### **Understanding lazy Instantiation of Singleton Pattern**

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

Let's see the simple example of singleton design pattern using lazy instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj;
3. **private** A(){}
5. **public** **static** A getA(){
6. **if** (obj == **null**){
7. **synchronized**(Singleton.**class**){
8. **if** (obj == **null**){
9. obj = **new** Singleton();//instance will be created at request time
10. }
11. }
12. }
13. **return** obj;
14. }
16. **public** **void** doSomething(){
17. //write your code
18. }
19. }

### **Significance of Classloader in Singleton Pattern**

#### If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.

### **Significance of Serialization in Singleton Pattern**

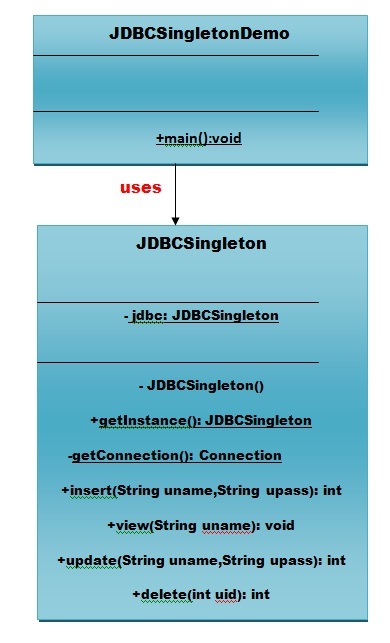
If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

1. **public** **class** A **implements** Serializable {
2. //your code of singleton
3. **protected** Object readResolve() {
4. **return** getA();
5. }
7. }

### **Understanding Real Example of Singleton Pattern**

* We are going to create a JDBCSingleton class. This JDBCSingleton class contains its constructor as private and a private static instance jdbc of itself.
* JDBCSingleton class provides a static method to get its static instance to the outside world. Now, JDBCSingletonDemo class will use JDBCSingleton class to get the JDBCSingleton object.



**Assumption:** you have created a table userdata that has three fields uid, uname and upassword in mysql database. Database name is ashwinirajput, username is root, password is ashwini.

*File: JDBCSingleton.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **import** java.sql.Connection;
5. **import** java.sql.DriverManager;
6. **import** java.sql.PreparedStatement;
7. **import** java.sql.ResultSet;
8. **import** java.sql.SQLException;
10. **class** JDBCSingleton {
11. //Step 1
12. // create a JDBCSingleton class.
13. //static member holds only one instance of the JDBCSingleton class.
15. **private** **static** JDBCSingleton jdbc;
17. //JDBCSingleton prevents the instantiation from any other class.
18. **private** JDBCSingleton() {  }
20. //Now we are providing gloabal point of access.
21. **public** **static** JDBCSingleton getInstance() {
22. **if** (jdbc==**null**)
23. {
24. jdbc=**new**  JDBCSingleton();
25. }
26. **return** jdbc;
27. }
29. // to get the connection from methods like insert, view etc.
30. **private** **static** Connection getConnection()**throws** ClassNotFoundException, SQLException
31. {
33. Connection con=**null**;
34. Class.forName("com.mysql.jdbc.Driver");
35. con= DriverManager.getConnection("jdbc:mysql://localhost:3306/ashwanirajput", "root", "ashwani");
36. **return** con;
38. }
40. //to insert the record into the database
41. **public** **int** insert(String name, String pass) **throws** SQLException
42. {
43. Connection c=**null**;
45. PreparedStatement ps=**null**;
47. **int** recordCounter=0;
49. **try** {
51. c=**this**.getConnection();
52. ps=c.prepareStatement("insert into userdata(uname,upassword)values(?,?)");
53. ps.setString(1, name);
54. ps.setString(2, pass);
55. recordCounter=ps.executeUpdate();
57. } **catch** (Exception e) { e.printStackTrace(); } **finally**{
58. **if** (ps!=**null**){
59. ps.close();
60. }**if**(c!=**null**){
61. c.close();
62. }
63. }
64. **return** recordCounter;
65. }
67. //to view the data from the database
68. **public**  **void** view(String name) **throws** SQLException
69. {
70. Connection con = **null**;
71. PreparedStatement ps = **null**;
72. ResultSet rs = **null**;
74. **try** {
76. con=**this**.getConnection();
77. ps=con.prepareStatement("select \* from userdata where uname=?");
78. ps.setString(1, name);
79. rs=ps.executeQuery();
80. **while** (rs.next()) {
81. System.out.println("Name= "+rs.getString(2)+"\t"+"Paasword= "+rs.getString(3));
83. }
85. } **catch** (Exception e) { System.out.println(e);}
86. **finally**{
87. **if**(rs!=**null**){
88. rs.close();
89. }**if** (ps!=**null**){
90. ps.close();
91. }**if**(con!=**null**){
92. con.close();
93. }
94. }
95. }
97. // to update the password for the given username
98. **public** **int** update(String name, String password) **throws** SQLException  {
99. Connection c=**null**;
100. PreparedStatement ps=**null**;
102. **int** recordCounter=0;
103. **try** {
104. c=**this**.getConnection();
105. ps=c.prepareStatement(" update userdata set upassword=? where uname='"+name+"' ");
106. ps.setString(1, password);
107. recordCounter=ps.executeUpdate();
108. } **catch** (Exception e) {  e.printStackTrace(); } **finally**{
110. **if** (ps!=**null**){
111. ps.close();
112. }**if**(c!=**null**){
113. c.close();
114. }
115. }
116. **return** recordCounter;
117. }
119. // to delete the data from the database
120. **public** **int** delete(**int** userid) **throws** SQLException{
121. Connection c=**null**;
122. PreparedStatement ps=**null**;
123. **int** recordCounter=0;
124. **try** {
125. c=**this**.getConnection();
126. ps=c.prepareStatement(" delete from userdata where uid='"+userid+"' ");
127. recordCounter=ps.executeUpdate();
128. } **catch** (Exception e) { e.printStackTrace(); }
129. **finally**{
130. **if** (ps!=**null**){
131. ps.close();
132. }**if**(c!=**null**){
133. c.close();
134. }
135. }
136. **return** recordCounter;
137. }
138. }// End of JDBCSingleton class

*File: JDBCSingletonDemo.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **import** java.sql.Connection;
5. **import** java.sql.DriverManager;
6. **import** java.sql.PreparedStatement;
7. **import** java.sql.ResultSet;
8. **import** java.sql.SQLException;
9. **class** JDBCSingletonDemo{
10. **static** **int** count=1;
11. **static** **int**  choice;
12. **public** **static** **void** main(String[] args) **throws** IOException {
14. JDBCSingleton jdbc= JDBCSingleton.getInstance();

17. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
18. **do**{
19. System.out.println("DATABASE OPERATIONS");
20. System.out.println(" --------------------- ");
21. System.out.println(" 1. Insertion ");
22. System.out.println(" 2. View      ");
23. System.out.println(" 3. Delete    ");
24. System.out.println(" 4. Update    ");
25. System.out.println(" 5. Exit      ");
27. System.out.print("\n");
28. System.out.print("Please enter the choice what you want to perform in the database: ");
30. choice=Integer.parseInt(br.readLine());
31. **switch**(choice) {
33. **case** 1:{
34. System.out.print("Enter the username you want to insert data into the database: ");
35. String username=br.readLine();
36. System.out.print("Enter the password you want to insert data into the database: ");
37. String password=br.readLine();
39. **try** {
40. **int** i= jdbc.insert(username, password);
41. **if** (i>0) {
42. System.out.println((count++) + " Data has been inserted successfully");
43. }**else**{
44. System.out.println("Data has not been inserted ");
45. }
47. } **catch** (Exception e) {
48. System.out.println(e);
49. }
51. System.out.println("Press Enter key to continue...");
52. System.in.read();
54. }//End of case 1
55. **break**;
56. **case** 2:{
57. System.out.print("Enter the username : ");
58. String username=br.readLine();
60. **try**  {
61. jdbc.view(username);
62. } **catch** (Exception e) {
63. System.out.println(e);
64. }
65. System.out.println("Press Enter key to continue...");
66. System.in.read();
68. }//End of case 2
69. **break**;
70. **case** 3:{
71. System.out.print("Enter the userid,  you want to delete: ");
72. **int** userid=Integer.parseInt(br.readLine());
74. **try** {
75. **int** i= jdbc.delete(userid);
76. **if** (i>0) {
77. System.out.println((count++) + " Data has been deleted successfully");
78. }**else**{
79. System.out.println("Data has not been deleted");
80. }
82. } **catch** (Exception e) {
83. System.out.println(e);
84. }
85. System.out.println("Press Enter key to continue...");
86. System.in.read();
88. }//End of case 3
89. **break**;
90. **case** 4:{
91. System.out.print("Enter the username,  you want to update: ");
92. String username=br.readLine();
93. System.out.print("Enter the new password ");
94. String password=br.readLine();
96. **try** {
97. **int** i= jdbc.update(username, password);
98. **if** (i>0) {
99. System.out.println((count++) + " Data has been updated successfully");
100. }
102. } **catch** (Exception e) {
103. System.out.println(e);
104. }
105. System.out.println("Press Enter key to continue...");
106. System.in.read();
108. }// end of case 4
109. **break**;
111. **default**:
112. **return**;
113. }
115. } **while** (choice!=4);
116. }
117. }

# Factory method:

Factory design pattern allows us to create the object without bothering on what is the actual class.

Being created.

The benefit is that the client code(calling code) can just say "give me an object that can do XYZ" without knowing what is the actual class that can do "XYZ".

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

## **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 circle

shape3.draw();

}

}

## **Step 5**

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Abstract Factory design pattern:

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

The **abstract factory design pattern** is merely an extension of the [factory method](http://www.dotnetlead.com/design-patterns/factory-method) pattern, which allows you to create objects without being concerned about the actual class of the objects being produced. The abstract factory pattern extends the [factory method](http://www.dotnetlead.com/design-patterns/factory-method)pattern by allowing more types of objects to be produced.

We can extend this [factory method](http://www.dotnetlead.com/design-patterns/factory-method) pattern to the abstract factory pattern by:

* Adding another product that the factories can produce.

#### Advantage of Abstract Factory Pattern

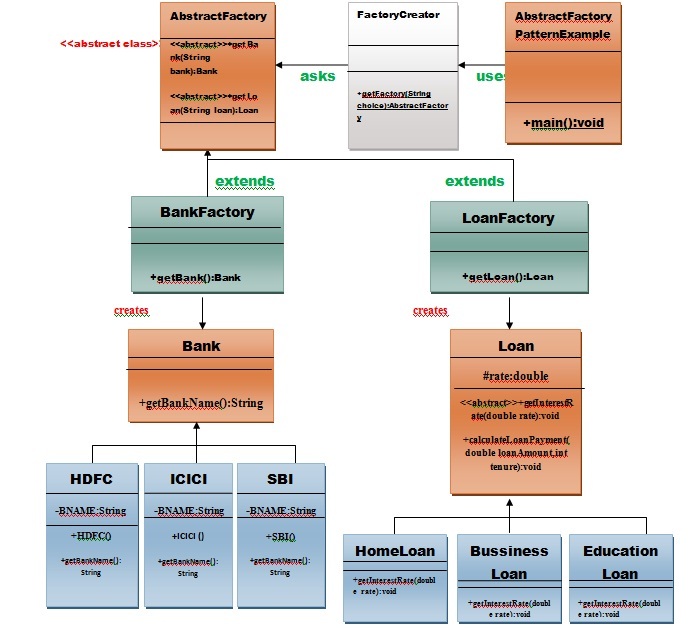
* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

#### Usage of Abstract Factory Pattern

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

### **UML for Abstract Factory Pattern**

* We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
* Then we will create **AbstractFactory** class as next step.
* Then after we will create concrete classes, **BankFactory,** and **LoanFactory** that will extends **AbstractFactory class**
* After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
* See the diagram carefully which is given below:



### **Example of Abstract Factory Pattern**

Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.

**Step 1:** Create a Bank interface

1. **import** java.io.\*;
2. **interface** Bank{
3. String getBankName();
4. }

**Step 2:** Create concrete classes that implement the Bank interface.

1. **class** HDFC **implements** Bank{
2. **private** **final** String BNAME;
3. **public** HDFC(){
4. BNAME="HDFC BANK";
5. }
6. **public** String getBankName() {
7. **return** BNAME;
8. }
9. }
10. **class** ICICI **implements** Bank{
11. **private** **final** String BNAME;
12. ICICI(){
13. BNAME="ICICI BANK";
14. }
15. **public** String getBankName() {
16. **return** BNAME;
17. }
18. }
19. **class** SBI **implements** Bank{
20. **private** **final** String BNAME;
21. **public** SBI(){
22. BNAME="SBI BANK";
23. }
24. **public** String getBankName(){
25. **return** BNAME;
26. }
27. }

**Step 3:** Create the Loan abstract class.

1. **abstract** **class** Loan{
2. **protected** **double** rate;
3. **abstract** **void** getInterestRate(**double** rate);
4. **public** **void** calculateLoanPayment(**double** loanamount, **int** years)
5. {
6. /\*
7. to calculate the monthly loan payment i.e. EMI
9. rate=annual interest rate/12\*100;
10. n=number of monthly installments;
11. 1year=12 months.
12. so, n=years\*12;
14. \*/
16. **double** EMI;
17. **int** n;
19. n=years\*12;
20. rate=rate/1200;
21. EMI=((rate\*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))\*loanamount;
23. System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");
24. }
25. }// end of the Loan abstract class.

**Step 4:** Create concrete classes that extend the Loan abstract class..

1. **class** HomeLoan **extends** Loan{
2. **public** **void** getInterestRate(**double** r){
3. rate=r;
4. }
5. }//End of the HomeLoan class.
6. **class** BussinessLoan **extends** Loan{
7. **public** **void** getInterestRate(**double** r){
8. rate=r;
9. }
11. }//End of the BusssinessLoan class.
12. **class** EducationLoan **extends** Loan{
13. **public** **void** getInterestRate(**double** r){
14. rate=r;
15. }
16. }//End of the EducationLoan class.

**Step 5:** Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

1. **abstract** **class** AbstractFactory{
2. **public** **abstract** Bank getBank(String bank);
3. **public** **abstract** Loan getLoan(String loan);
4. }

**Step 6:** Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

1. **class** BankFactory **extends** AbstractFactory{
2. **public** Bank getBank(String bank){
3. **if**(bank == **null**){
4. **return** **null**;
5. }
6. **if**(bank.equalsIgnoreCase("HDFC")){
7. **return** **new** HDFC();
8. } **else** **if**(bank.equalsIgnoreCase("ICICI")){
9. **return** **new** ICICI();
10. } **else** **if**(bank.equalsIgnoreCase("SBI")){
11. **return** **new** SBI();
12. }
13. **return** **null**;
14. }
15. **public** Loan getLoan(String loan) {
16. **return** **null**;
17. }
18. }//End of the BankFactory class.
19. **class** LoanFactory **extends** AbstractFactory{
20. **public** Bank getBank(String bank){
21. **return** **null**;
22. }
24. **public** Loan getLoan(String loan){
25. **if**(loan == **null**){
26. **return** **null**;
27. }
28. **if**(loan.equalsIgnoreCase("Home")){
29. **return** **new** HomeLoan();
30. } **else** **if**(loan.equalsIgnoreCase("Business")){
31. **return** **new** BussinessLoan();
32. } **else** **if**(loan.equalsIgnoreCase("Education")){
33. **return** **new** EducationLoan();
34. }
35. **return** **null**;
36. }
38. }

**Step 7:** Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

1. **class** FactoryCreator {
2. **public** **static** AbstractFactory getFactory(String choice){
3. **if**(choice.equalsIgnoreCase("Bank")){
4. **return** **new** BankFactory();
5. } **else** **if**(choice.equalsIgnoreCase("Loan")){
6. **return** **new** LoanFactory();
7. }
8. **return** **null**;
9. }
10. }//End of the FactoryCreator.

**Step 8:** Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

1. **import** java.io.\*;
2. **class** AbstractFactoryPatternExample {
3. **public** **static** **void** main(String args[])**throws** IOException {
5. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
7. System.out.print("Enter the name of Bank from where you want to take loan amount: ");
8. String bankName=br.readLine();
10. System.out.print("\n");
11. System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");
13. String loanName=br.readLine();
14. AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
15. Bank b=bankFactory.getBank(bankName);
17. System.out.print("\n");
18. System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");
20. **double** rate=Double.parseDouble(br.readLine());
21. System.out.print("\n");
22. System.out.print("Enter the loan amount you want to take: ");
24. **double** loanAmount=Double.parseDouble(br.readLine());
25. System.out.print("\n");
26. System.out.print("Enter the number of years to pay your entire loan amount: ");
27. **int** years=Integer.parseInt(br.readLine());
29. System.out.print("\n");
30. System.out.println("you are taking the loan from "+ b.getBankName());
32. AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
33. Loan l=loanFactory.getLoan(loanName);
34. l.getInterestRate(rate);
35. l.calculateLoanPayment(loanAmount,years);
36. }
37. }//End of the  AbstractFactoryPatternExample

Builder design pattern:

The **builder design pattern** allows you to create a general guideline on how to create an object, then have different implementations on how to build parts of the object.

There are 2 principles in the builder pattern, let's use an example of building an airplane to demonstrate the principles:  
  
The first principle is the general guideline that must be followed when building an object. For example, in building an airplane the body must be constructed before the wings. This general guideline must be followed regardless what types of airplane you are building.   
  
The second principle are the different specifications on building the parts of the airplane. When building a jet airplane the body must be built differently than a propeller airplane. These specifications are included in the pattern.

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

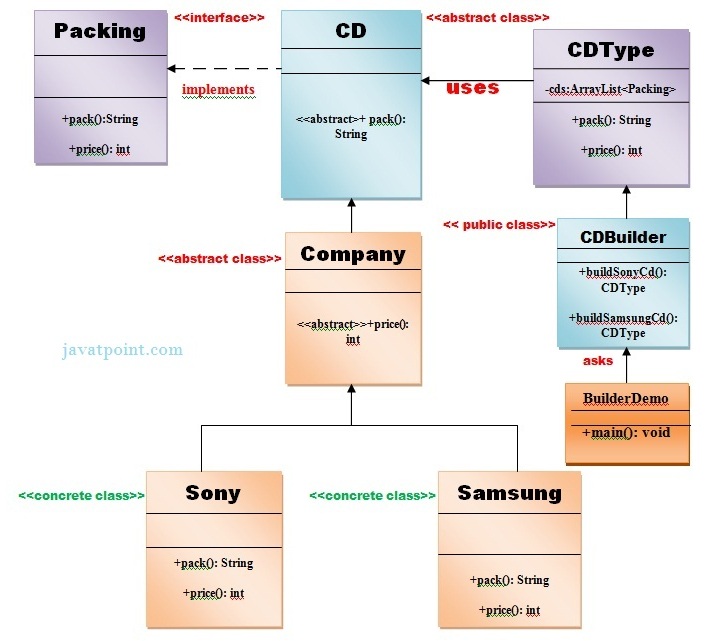
It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

#### Advantage of Builder Design Pattern

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

#### UML for Builder Pattern Example



### **Example of Builder Design Pattern**

To create simple example of builder design pattern, you need to follow 6 following steps.

1. Create Packing interface
2. Create 2 abstract classes CD and Company
3. Create 2 implementation classes of Company: Sony and Samsung
4. Create the CDType class
5. Create the CDBuilder class
6. Create the BuilderDemo class

#### 1) Create Packing interface

*File: Packing.java*

1. **public** **interface** Packing {
2. **public** String pack();
3. **public** **int** price();
4. }

#### 2) Create 2 abstract classes CD and Company

Create an abstract class CD which will implement Packing interface.

*File: CD.java*

1. **public** **abstract** **class** CD **implements** Packing{
2. **public** **abstract** String pack();
3. }

*File: Company.java*

1. **public** **abstract** **class** Company **extends** CD{
2. **public** **abstract** **int** price();
3. }

#### 3) Create 2 implementation classes of Company: Sony and Samsung

*File: Sony.java*

1. **public** **class** Sony **extends** Company{
2. @Override
3. **public** **int** price(){
4. **return** 20;
5. }
6. @Override
7. **public** String pack(){
8. **return** "Sony CD";
9. }
10. }//End of the Sony class.

*File: Samsung.java*

1. **public** **class** Samsung **extends** Company {
2. @Override
3. **public** **int** price(){
4. **return** 15;
5. }
6. @Override
7. **public** String pack(){
8. **return** "Samsung CD";
9. }
10. }//End of the Samsung class.

#### 4) Create the CDType class

*File: CDType.java*

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** CDType {
4. **private** List<Packing> items=**new** ArrayList<Packing>();
5. **public** **void** addItem(Packing packs) {
6. items.add(packs);
7. }
8. **public** **void** getCost(){
9. **for** (Packing packs : items) {
10. packs.price();
11. }
12. }
13. **public** **void** showItems(){
14. **for** (Packing packing : items){
15. System.out.print("CD name : "+packing.pack());
16. System.out.println(", Price : "+packing.price());
17. }
18. }
19. }//End of the CDType class.

#### 5) Create the CDBuilder class

*File: CDBuilder.java*

1. **public** **class** CDBuilder {
2. **public** CDType buildSonyCD(){
3. CDType cds=**new** CDType();
4. cds.addItem(**new** Sony());
5. **return** cds;
6. }
7. **public** CDType buildSamsungCD(){
8. CDType cds=**new** CDType();
9. cds.addItem(**new** Samsung());
10. **return** cds;
11. }
12. }// End of the CDBuilder class.

#### 6) Create the BuilderDemo class

*File: BuilderDemo.java*

1. **public** **class** BuilderDemo{
2. **public** **static** **void** main(String args[]){
3. CDBuilder cdBuilder=**new** CDBuilder();
4. CDType cdType1=cdBuilder.buildSonyCD();
5. cdType1.showItems();
7. CDType cdType2=cdBuilder.buildSamsungCD();
8. cdType2.showItems();
9. }
10. }

[download this builder pattern example](https://www.javatpoint.com/designpattern/designpatternexample/builder1.zip)

#### Output of the above example

1. CD name : Sony CD, Price : 20
2. CD name : Samsung CD, Price : 15

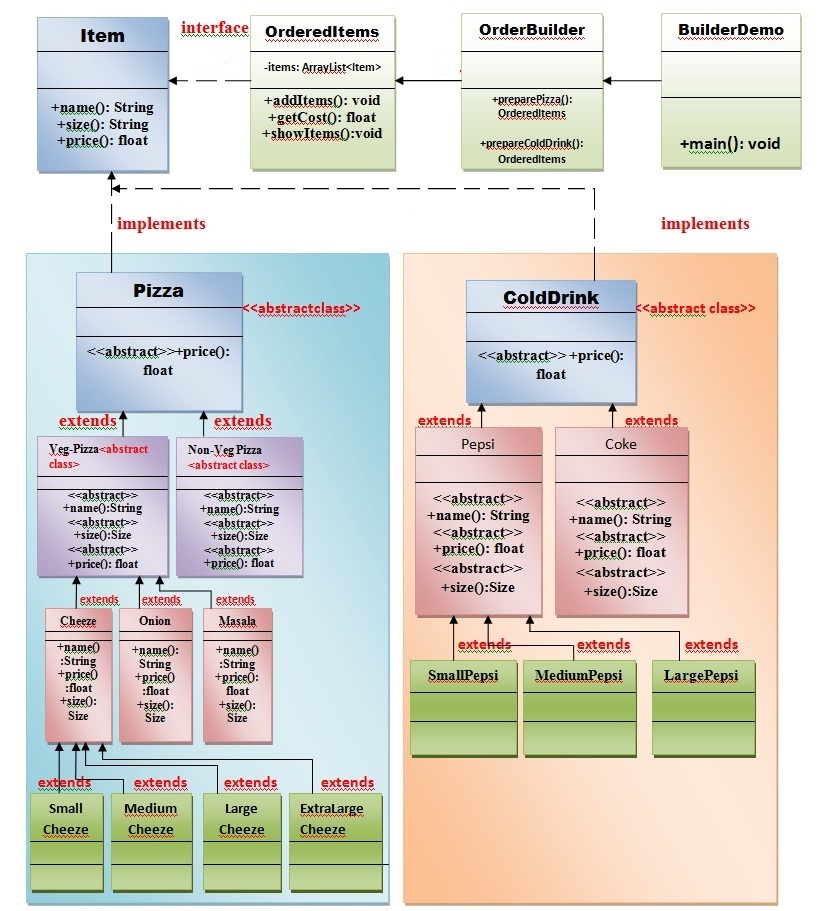
### **Another Real world example of Builder Pattern**

#### UML for Builder Pattern:

We are considering a business case of **pizza-hut** where we can get different varieties of pizza and cold-drink.

**Pizza** can be either a Veg pizza or Non-Veg pizza of several types (like cheese pizza, onion pizza, masala-pizza etc) and will be of 4 sizes i.e. small, medium, large, extra-large.

**Cold-drink** can be of several types (like Pepsi, Coke, Dew, Sprite, Fanta, Maaza, Limca, Thums-up etc.) and will be of 3 sizes small, medium, large.



#### Real world example of builder pattern

Let's see the step by step real world example of Builder Design Pattern.

Step 1:**Create an interface Item that represents the Pizza and Cold-drink.**

*File: Item.java*

1. **public**  **interface**  Item
2. {
3. **public** String name();
4. **public** String size();
5. **public** **float** price();
6. }// End of the interface Item.

Step 2:**Create an abstract class Pizza that will implement to the interface Item.**

*File: Pizza.java*

1. **public** **abstract** **class** Pizza **implements** Item{
2. @Override
3. **public** **abstract** **float** price();
4. }

Step 3:**Create an abstract class ColdDrink that will implement to the interface Item.**

*File: ColdDrink.java*

1. **public** **abstract** **class** ColdDrink **implements** Item{
2. @Override
3. **public** **abstract** **float** price();

Step 4:**Create an abstract class VegPizza that will extend to the abstract class Pizza.**

*File: VegPizza.java*

1. **public** **abstract** **class** VegPizza **extends** Pizza{
2. @Override
3. **public** **abstract** **float** price();
4. @Override
5. **public** **abstract**  String name();
6. @Override
7. **public** **abstract**  String size();
8. }// End of the abstract class VegPizza.

Step 5:**Create an abstract class NonVegPizza that will extend to the abstract class Pizza.**

*File: NonVegPizza.java*

1. **public** **abstract** **class** NonVegPizza **extends** Pizza{
2. @Override
3. **public** **abstract** **float** price();
4. @Override
5. **public** **abstract** String name();
6. @Override
7. **public** **abstract** String size();
8. }// End of the abstract class NonVegPizza.

Step 6:**Now, create concrete sub-classes SmallCheezePizza, MediumCheezePizza, LargeCheezePizza, ExtraLargeCheezePizza that will extend to the abstract class VegPizza.**

*File: SmallCheezePizza.java*

1. **public** **class** SmallCheezePizza **extends** VegPizza{
2. @Override
3. **public** **float** price() {
4. **return** 170.0f;
5. }
6. @Override
7. **public** String name() {
8. **return** "Cheeze Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return** "Small size";
13. }
14. }// End of the SmallCheezePizza class.

*File: MediumCheezePizza.java*

1. **public** **class** MediumCheezePizza **extends** VegPizza{
2. @Override
3. **public** **float** price() {
4. **return**  220.f;
5. }
6. @Override
7. **public** String name() {
8. **return** "Cheeze Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return** "Medium Size";
13. }
14. }// End of the MediumCheezePizza class.
15. </textaera></div>
17. <div id="filename">File: LargeCheezePizza.java</div>
18. <div **class**="codeblock"><textarea  name="code" **class**="java">
19. **public** **class** LargeCheezePizza **extends** VegPizza{
20. @Override
21. **public** **float** price() {
22. **return** 260.0f;
23. }
24. @Override
25. **public** String name() {
26. **return** "Cheeze Pizza";
27. }
28. @Override
29. **public** String size() {
30. **return** "Large Size";
31. }
32. }// End of the LargeCheezePizza class.

*File: ExtraLargeCheezePizza.java*

1. **public** **class** ExtraLargeCheezePizza **extends** VegPizza{
2. @Override
3. **public** **float** price() {
4. **return** 300.f;
5. }
6. @Override
7. **public** String name() {
8. **return**  "Cheeze Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return** "Extra-Large Size";
13. }
14. }// End of the ExtraLargeCheezePizza class.

Step 7:**Now, similarly create concrete sub-classes SmallOnionPizza, MediumOnionPizza, LargeOnionPizza, ExtraLargeOnionPizza that will extend to the abstract class VegPizza.**

*File: SmallOnionPizza.java*

1. **public** **class** SmallOnionPizza **extends** VegPizza {
2. @Override
3. **public** **float** price() {
4. **return** 120.0f;
5. }
6. @Override
7. **public** String name() {
8. **return**  "Onion Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return**  "Small Size";
13. }
14. }// End of the SmallOnionPizza class.

*File: MediumOnionPizza.java*

1. **public** **class** MediumOnionPizza **extends** VegPizza {
2. @Override
3. **public** **float** price() {
4. **return** 150.0f;
5. }
6. @Override
7. **public** String name() {
8. **return**  "Onion Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return**  "Medium Size";
13. }
14. }// End of the MediumOnionPizza class.

*File: LargeOnionPizza.java*

1. **public** **class** LargeOnionPizza **extends**  VegPizza{
2. @Override
3. **public** **float** price() {
4. **return** 180.0f;
5. }
6. @Override
7. **public** String name() {
8. **return** "Onion Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return**  "Large size";
13. }
14. }// End of the LargeOnionPizza class.

*File: ExtraLargeOnionPizza.java*

1. **public** **class** ExtraLargeOnionPizza **extends** VegPizza {
2. @Override
3. **public** **float** price() {
4. **return** 200.0f;
5. }
6. @Override
7. **public** String name() {
8. **return**  "Onion Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return**  "Extra-Large Size";
13. }
14. }// End of the ExtraLargeOnionPizza class

Step 8:**Now, similarly create concrete sub-classes SmallMasalaPizza, MediumMasalaPizza, LargeMasalaPizza, ExtraLargeMasalaPizza that will extend to the abstract class VegPizza.**

*File: SmallMasalaPizza.java*

1. **public** **class** SmallMasalaPizza **extends** VegPizza{
2. @Override
3. **public** **float** price() {
4. **return** 100.0f;
5. }
6. @Override
7. **public** String name() {
8. **return**  "Masala Pizza";
9. }
10. @Override
11. **public** String size() {
12. **return**  "Samll Size";
13. }
14. }// End of the SmallMasalaPizza class

*File: MediumMasalaPizza.java*

1. **public** **class** MediumMasalaPizza **extends** VegPizza {
3. @Override
4. **public** **float** price() {
5. **return** 120.0f;
6. }
8. @Override
9. **public** String name() {
11. **return**  "Masala Pizza";
13. }
15. @Override
16. **public** String size() {
17. **return**  "Medium Size";
18. }

*File: LargeMasalaPizza.java*

1. **public** **class** LargeMasalaPizza **extends**  VegPizza{
2. @Override
3. **public** **float** price() {
4. **return** 150.0f;
5. }
7. @Override
8. **public** String name() {
10. **return**  "Masala Pizza";
12. }
14. @Override
15. **public** String size() {
16. **return**  "Large Size";
17. }
18. } //End of the LargeMasalaPizza class

*File: ExtraLargeMasalaPizza.java*

1. **public** **class** ExtraLargeMasalaPizza **extends** VegPizza {
2. @Override
3. **public** **float** price() {
4. **return** 180.0f;
5. }
7. @Override
8. **public** String name() {
10. **return**  "Masala Pizza";
12. }
14. @Override
15. **public** String size() {
16. **return**  "Extra-Large Size";
17. }
18. }// End of the ExtraLargeMasalaPizza class

Step 9:**Now, create concrete sub-classes SmallNonVegPizza, MediumNonVegPizza, LargeNonVegPizza, ExtraLargeNonVegPizza that will extend to the abstract class NonVegPizza.**

*File: SmallNonVegPizza.java*

1. **public** **class** SmallNonVegPizza **extends** NonVegPizza {
3. @Override
4. **public** **float** price() {
5. **return** 180.0f;
6. }
8. @Override
9. **public** String name() {
10. **return** "Non-Veg Pizza";
11. }
13. @Override
14. **public** String size() {
15. **return** "Samll Size";
16. }
18. }// End of the SmallNonVegPizza class

*File: MediumNonVegPizza.java*

1. **public** **class** MediumNonVegPizza **extends** NonVegPizza{
3. @Override
4. **public** **float** price() {
5. **return** 200.0f;
6. }
8. @Override
9. **public** String name() {
10. **return** "Non-Veg Pizza";
11. }
13. @Override
14. **public** String size() {
15. **return** "Medium Size";
16. }

*File: LargeNonVegPizza.java*

1. **public** **class** LargeNonVegPizza **extends** NonVegPizza{
3. @Override
4. **public** **float** price() {
5. **return** 220.0f;
6. }
8. @Override
9. **public** String name() {
10. **return** "Non-Veg Pizza";
11. }
13. @Override
14. **public** String size() {
15. **return** "Large Size";
16. }
18. }// End of the LargeNonVegPizza class

*File: ExtraLargeNonVegPizza.java*

1. **public** **class** ExtraLargeNonVegPizza **extends** NonVegPizza {
2. @Override
3. **public** **float** price() {
4. **return** 250.0f;
5. }
7. @Override
8. **public** String name() {
9. **return** "Non-Veg Pizza";
10. }
12. @Override
13. **public** String size() {
14. **return** "Extra-Large Size";
15. }
17. }
19. // End of the ExtraLargeNonVegPizza class

Step 10:**Now, create two abstract classes Pepsi and Coke that will extend abstract class ColdDrink.**

*File: Pepsi.java*

1. **public** **abstract** **class** Pepsi **extends** ColdDrink {
3. @Override
4. **public** **abstract**  String name();
6. @Override
7. **public** **abstract**  String size();
9. @Override
10. **public** **abstract**  **float** price();
12. }// End of the Pepsi class

*File: Coke.java*

1. **public** **abstract** **class** Coke  **extends** ColdDrink {
3. @Override
4. **public** **abstract**  String name();
6. @Override
7. **public** **abstract**  String size();
9. @Override
10. **public** **abstract**  **float** price();
12. }// End of the Coke class
14. </textaea></div>
16. <p>Step 11:<b>Now, create concrete sub-classes SmallPepsi, MediumPepsi, LargePepsi that will extend to the **abstract** **class** Pepsi.</b></p>
17. <div id="filename">File: SmallPepsi.java</div>
18. <div **class**="codeblock"><textarea  name="code" **class**="java">
19. **public** **class** SmallPepsi  **extends** Pepsi{
21. @Override
22. **public** String name() {
23. **return** "300 ml Pepsi";
24. }
26. @Override
27. **public** **float** price() {
28. **return** 25.0f;
29. }
31. @Override
32. **public** String size() {
33. **return** "Small Size";
34. }
35. }// End of the SmallPepsi class

*File: MediumPepsi.java*

1. **public** **class** MediumPepsi **extends** Pepsi {
3. @Override
4. **public** String name() {
5. **return** "500 ml Pepsi";
6. }
8. @Override
9. **public** String size() {
10. **return** "Medium Size";
11. }
13. @Override
14. **public** **float** price() {
15. **return** 35.0f;
16. }
17. }// End of the MediumPepsi class

*File: LargePepsi.java*

1. **public** **class** LargePepsi **extends** Pepsi{
2. @Override
3. **public** String name() {
4. **return** "750 ml Pepsi";
5. }
7. @Override
8. **public** String size() {
9. **return** "Large Size";
10. }
12. @Override
13. **public** **float** price() {
14. **return** 50.0f;
15. }
16. }// End of the LargePepsi class

Step 12:**Now, create concrete sub-classes SmallCoke, MediumCoke, LargeCoke that will extend to the abstract class Coke.**

*File: SmallCoke.java*

1. **public** **class** SmallCoke **extends** Coke{
3. @Override
4. **public** String name() {
5. **return** "300 ml Coke";
6. }
8. @Override
9. **public** String size() {
11. **return** "Small Size";
12. }
14. @Override
15. **public** **float** price() {
17. **return**  25.0f;
18. }
19. }// End of the SmallCoke class

*File: MediumCoke.java*

1. **public** **class** MediumCoke **extends** Coke{
3. @Override
4. **public** String name() {
5. **return** "500 ml Coke";
6. }
8. @Override
9. **public** String size() {
11. **return** "Medium Size";
12. }
14. @Override
15. **public** **float** price() {
17. **return**  35.0f;
18. }
19. }// End of the MediumCoke class

*File: LargeCoke.java*

1. **public** **class** LargeCoke **extends** Coke {
2. @Override
3. **public** String name() {
4. **return** "750 ml Coke";
5. }
7. @Override
8. **public** String size() {
10. **return** "Large Size";
11. }
13. @Override
14. **public** **float** price() {
16. **return**  50.0f;
17. }
18. }// End of the LargeCoke class
20. </textrea></div>
22. <p>Step 13:<b>Create an OrderedItems **class** that are having Item objects defined above.</b></p>
23. <div id="filename">File: OrderedItems.java</div>
24. <div **class**="codeblock"><textarea  name="code" **class**="java">
25. **import** java.util.ArrayList;
26. **import** java.util.List;
27. **public** **class** OrderedItems {
29. List<Item> items=**new** ArrayList<Item>();
31. **public** **void** addItems(Item item){
33. items.add(item);
34. }
35. **public** **float** getCost(){
37. **float** cost=0.0f;
38. **for** (Item item : items) {
39. cost+=item.price();
40. }
41. **return** cost;
42. }
43. **public** **void** showItems(){
45. **for** (Item item : items) {
46. System.out.println("Item is:" +item.name());
47. System.out.println("Size is:" +item.size());
48. System.out.println("Price is: " +item.price());
50. }
51. }
53. }// End of the OrderedItems class

Step 14:**Create an OrderBuilder class that will be responsible to create the objects of OrderedItems class.**

*File: OrdereBuilder.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **public** **class** OrderBuilder {
5. **public** OrderedItems preparePizza() **throws** IOException{
7. OrderedItems itemsOrder=**new** OrderedItems();
8. BufferedReader br =**new** BufferedReader(**new** InputStreamReader(System.in));
10. System.out.println(" Enter the choice of Pizza ");
11. System.out.println("============================");
12. System.out.println("        1. Veg-Pizza       ");
13. System.out.println("        2. Non-Veg Pizza   ");
14. System.out.println("        3. Exit            ");
15. System.out.println("============================");
17. **int** pizzaandcolddrinkchoice=Integer.parseInt(br.readLine());
18. **switch**(pizzaandcolddrinkchoice)
19. {
20. **case** 1:{
22. System.out.println("You ordered Veg Pizza");
23. System.out.println("\n\n");
24. System.out.println(" Enter the types of Veg-Pizza ");
25. System.out.println("------------------------------");
26. System.out.println("        1.Cheeze Pizza        ");
27. System.out.println("        2.Onion Pizza        ");
28. System.out.println("        3.Masala Pizza        ");
29. System.out.println("        4.Exit            ");
30. System.out.println("------------------------------");
31. **int** vegpizzachoice=Integer.parseInt(br.readLine());
32. **switch**(vegpizzachoice)
33. {
34. **case** 1:
35. {
36. System.out.println("You ordered Cheeze Pizza");
38. System.out.println("Enter the cheeze pizza size");
39. System.out.println("------------------------------------");
40. System.out.println("    1. Small Cheeze Pizza ");
41. System.out.println("    2. Medium Cheeze Pizza ");
42. System.out.println("    3. Large Cheeze Pizza ");
43. System.out.println("    4. Extra-Large Cheeze Pizza ");
44. System.out.println("------------------------------------");
45. **int** cheezepizzasize=Integer.parseInt(br.readLine());
46. **switch**(cheezepizzasize)
47. {
48. **case** 1:
49. itemsOrder.addItems(**new** SmallCheezePizza());
50. **break**;
51. **case** 2:
52. itemsOrder.addItems(**new** MediumCheezePizza());
53. **break**;
54. **case** 3:
55. itemsOrder.addItems(**new** LargeCheezePizza());
56. **break**;
57. **case** 4:
58. itemsOrder.addItems(**new** ExtraLargeCheezePizza());
59. **break**;
60. **case** 2:
61. {
62. System.out.println("You ordered Onion Pizza");
63. System.out.println("Enter the Onion pizza size");
64. System.out.println("----------------------------------");
65. System.out.println("    1. Small Onion Pizza ");
66. System.out.println("    2. Medium Onion Pizza ");
67. System.out.println("    3. Large Onion Pizza ");
68. System.out.println("    4. Extra-Large Onion Pizza ");
69. System.out.println("----------------------------------");
70. **int** onionpizzasize=Integer.parseInt(br.readLine());
71. **switch**(onionpizzasize)
72. {
73. **case** 1:
74. itemsOrder.addItems(**new** SmallOnionPizza());
75. **break**;
77. **case** 2:
78. itemsOrder.addItems(**new** MediumOnionPizza());
79. **break**;
81. **case** 3:
82. itemsOrder.addItems(**new** LargeOnionPizza());
83. **break**;
85. **case** 4:
86. itemsOrder.addItems(**new** ExtraLargeOnionPizza());
87. **break**;
89. }
90. }
91. **break**;
92. **case** 3:
93. {
94. System.out.println("You ordered Masala Pizza");
95. System.out.println("Enter the Masala pizza size");
96. System.out.println("------------------------------------");
97. System.out.println("    1. Small Masala Pizza ");
98. System.out.println("    2. Medium Masala Pizza ");
99. System.out.println("    3. Large Masala Pizza ");
100. System.out.println("    4. Extra-Large Masala Pizza ");
101. System.out.println("------------------------------------");
102. **int** masalapizzasize=Integer.parseInt(br.readLine());
103. **switch**(masalapizzasize)
104. {
105. **case** 1:
106. itemsOrder.addItems(**new** SmallMasalaPizza());
107. **break**;
109. **case** 2:
110. itemsOrder.addItems(**new** MediumMasalaPizza());
111. **break**;
113. **case** 3:
114. itemsOrder.addItems(**new** LargeMasalaPizza());
115. **break**;
117. **case** 4:
118. itemsOrder.addItems(**new** ExtraLargeMasalaPizza());
119. **break**;
121. }
123. }
124. **break**;
126. }
128. }
129. **break**;// Veg- pizza choice completed.
131. **case** 2:
132. {
133. System.out.println("You ordered Non-Veg Pizza");
134. System.out.println("\n\n");
136. System.out.println("Enter the Non-Veg pizza size");
137. System.out.println("------------------------------------");
138. System.out.println("    1. Small Non-Veg  Pizza ");
139. System.out.println("    2. Medium Non-Veg  Pizza ");
140. System.out.println("    3. Large Non-Veg  Pizza ");
141. System.out.println("    4. Extra-Large Non-Veg  Pizza ");
142. System.out.println("------------------------------------");

145. **int** nonvegpizzasize=Integer.parseInt(br.readLine());
147. **switch**(nonvegpizzasize)
148. {
150. **case** 1:
151. itemsOrder.addItems(**new** SmallNonVegPizza());
152. **break**;
154. **case** 2:
155. itemsOrder.addItems(**new** MediumNonVegPizza());
156. **break**;
158. **case** 3:
159. itemsOrder.addItems(**new** LargeNonVegPizza());
160. **break**;
162. **case** 4:
163. itemsOrder.addItems(**new** ExtraLargeNonVegPizza());
164. **break**;
165. }
167. }
168. **break**;
169. **default**:
170. {
171. **break**;
173. }
175. }//end of main Switch
177. //continued?..
178. System.out.println(" Enter the choice of ColdDrink ");
179. System.out.println("============================");
180. System.out.println("        1. Pepsi            ");
181. System.out.println("        2. Coke             ");
182. System.out.println("        3. Exit             ");
183. System.out.println("============================");
184. **int** coldDrink=Integer.parseInt(br.readLine());
185. **switch** (coldDrink)
186. {
187. **case** 1:
188. {
189. System.out.println("You ordered Pepsi ");
190. System.out.println("Enter the  Pepsi Size ");
191. System.out.println("------------------------");
192. System.out.println("    1. Small Pepsi ");
193. System.out.println("    2. Medium Pepsi ");
194. System.out.println("    3. Large Pepsi ");
195. System.out.println("------------------------");
196. **int** pepsisize=Integer.parseInt(br.readLine());
197. **switch**(pepsisize)
198. {
199. **case** 1:
200. itemsOrder.addItems(**new** SmallPepsi());
201. **break**;
203. **case** 2:
204. itemsOrder.addItems(**new** MediumPepsi());
205. **break**;
207. **case** 3:
208. itemsOrder.addItems(**new** LargePepsi());
209. **break**;
211. }
212. }
213. **break**;
214. **case** 2:
215. {
216. System.out.println("You ordered Coke");
217. System.out.println("Enter the Coke Size");
218. System.out.println("------------------------");
219. System.out.println("    1. Small Coke ");
220. System.out.println("    2. Medium Coke  ");
221. System.out.println("    3. Large Coke  ");
222. System.out.println("    4. Extra-Large Coke ");
223. System.out.println("------------------------");
225. **int** cokesize=Integer.parseInt(br.readLine());
226. **switch**(cokesize)
227. {
228. **case** 1:
229. itemsOrder.addItems(**new** SmallCoke());
230. **break**;
232. **case** 2:
233. itemsOrder.addItems(**new** MediumCoke());
234. **break**;
236. **case** 3:
237. itemsOrder.addItems(**new** LargeCoke());
238. **break**;

241. }
243. }
244. **break**;
245. **default**:
246. {
247. **break**;
249. }
251. }//End of the Cold-Drink switch
252. **return** itemsOrder;
254. } //End of the preparePizza() method

Step 15:**Create a BuilderDemo class that will use the OrderBuilder class.**

*File: Prototype.java*

1. **import** java.io.IOException;
2. **public** **class** BuilderDemo {
4. **public** **static** **void** main(String[] args) **throws** IOException {
5. // TODO code application logic here
7. OrderBuilder builder=**new** OrderBuilder();
9. OrderedItems orderedItems=builder.preparePizza();
11. orderedItems.showItems();
13. System.out.println("\n");
14. System.out.println("Total Cost : "+ orderedItems.getCost());
16. }
17. }// End of the BuilderDemo class

Prototype design pattern:

The **prototype design pattern** allows you to avoid expensive initialization routines when you construct objects that are very similar. The goal is to minimize the amount of work needed in creating new objects when the initialization routines are expensive. For example, if the initialization routine requires database queries, file lookups, or service calls and you already have other objects in the system that are very similar to the object you are constructing, then the prototype pattern may help you to avoid those expensive initializations.

The concept of the prototype pattern is similar to a word document template (prototype), where you create the template only once, then all the objects that you create afterwards uses the template as the starting point to avoid repeating the work again.

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Advantage of Prototype Pattern

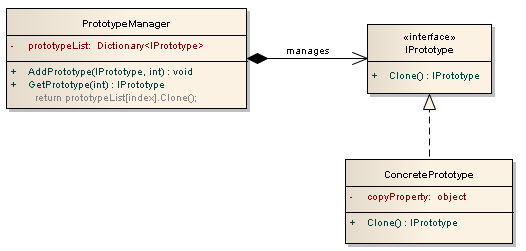
The main advantages of prototype pattern are as follows:

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

Usage of Prototype Pattern

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.

Let's look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the prototype pattern first, then we will look an example to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Prototype Design Pattern:

[](http://www.dotnetlead.com/design-patterns/prototype/prototype1.PNG?attredirects=0)

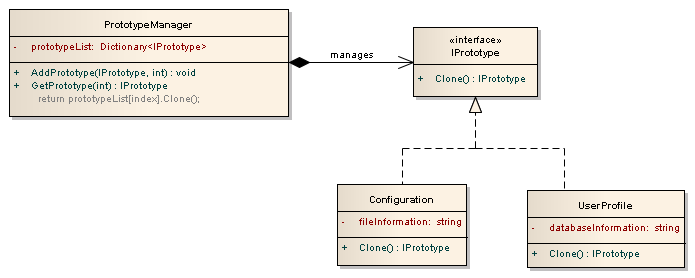
* The *PrototypeManager*class is just a manager class that is used to add and retrieve prototypes by an index number, it has the following variable and methods:
  + *prototypeList* variable is the collection that stores all the prototypes
  + *AddPrototype* method allows you to add a prototype to the collection and assigning it an index number
  + *GetPrototype* method allows you to retrieve a prototype from the collection using an index number
* The *IPrototype*interface specifies the methods that all prototype classes must implement. It has the *Clone* method that returns an *IPrototype*interface.
* The *ConcretePrototype*class is the actual prototype class, it implements the *IPrototype*interface and has the following property and method:
  + *copyProperty* variable holds the information that is prepopulated. If the variable value is changed then the new instances created will have the new value.
  + *Clone*method will make a copy of itself and return it. If the *copyProperty*is a value type (such as int or string) then you can just use shallow copy. If the *copyProperty*is a reference type (such as an object that contains other objects) then you will need to make deep copy of the variable.

The key to the prototype pattern is that you will create your first object with the expensive initializations, then store the values as a prototype in the repository. When you need create the same object again you can just get the copy of the prototype from the repository with all the values prepopulated.

While there are many supported ways of making shallow or deep copy of objects in various programming languages, for the purpose of understanding the prototype pattern we will not dig into such details but simply demonstrate the core concept of the prototype pattern.

A comparison between the prototype pattern and the [flyweight pattern](http://www.dotnetlead.com/design-patterns/flyweight) shows some similarities in the [UML](http://www.dotnetlead.com/UML-Quick-Reference), in that both uses a manager to store and retrieve the objects in the collection. But there is a clear difference between the two. The prototype pattern is used to create new objects that are similar in nature (hence it's a creational pattern), while the [flyweight pattern](http://www.dotnetlead.com/design-patterns/flyweight)is used to allow the application to point to the same instance of the object to save memory (hence it's a structural pattern).

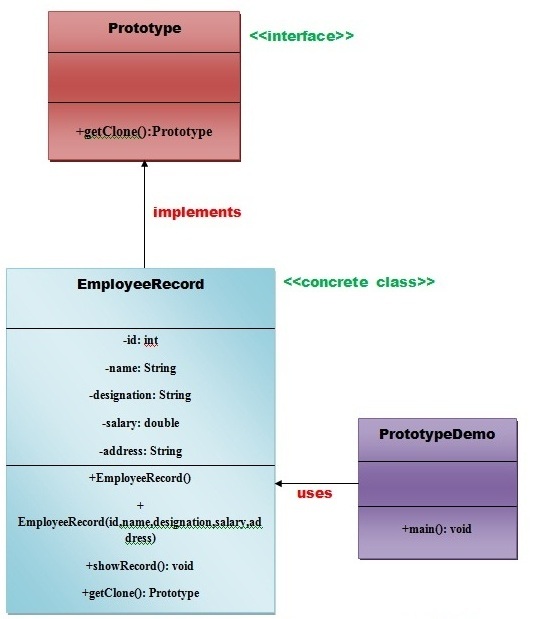
Let's see an example. In our example we will often need to create *Configuration* and *UserProfile* objects in the application, and both objects stores information that are resource intensive and takes a while to populate. The [UML](http://www.dotnetlead.com/UML-Quick-Reference) for the example is shown below:

[](http://www.dotnetlead.com/design-patterns/prototype/prototype2.PNG?attredirects=0)

The *Configuration*class has the *fileInformation* variable that takes a while to create. The same goes for the *UserProfile* class which has the *databaseInformation*variable that is resource intensive to create.

Notice that it takes a while to create the prototype class for the first time, but subsequent objects created afterwards takes no time at all:

UML for Prototype Pattern



* We are going to create **an interface Prototype** that contains a method **getClone()** of **Prototype type.**
* Then, we create **a concrete class EmployeeRecord** which implements **Prototype interface** that does the cloning of EmployeeRecord object.
* **PrototypeDemo class** will uses this concrete class **EmployeeRecord.**

Example of Prototype Design Pattern

Let's see the example of prototype design pattern.

*File: Prototype.java*

1. **interface** Prototype {
3. **public** Prototype getClone();
5. }//End of Prototype interface.

*File: EmployeeRecord.java*

1. **class** EmployeeRecord **implements** Prototype{
3. **private** **int** id;
4. **private** String name, designation;
5. **private** **double** salary;
6. **private** String address;
8. **public** EmployeeRecord(){
9. System.out.println("   Employee Records of Oracle Corporation ");
10. System.out.println("---------------------------------------------");
11. System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Eaddress");
13. }
15. **public**  EmployeeRecord(**int** id, String name, String designation, **double** salary, String address) {
17. **this**();
18. **this**.id = id;
19. **this**.name = name;
20. **this**.designation = designation;
21. **this**.salary = salary;
22. **this**.address = address;
23. }
25. **public** **void** showRecord(){
27. System.out.println(id+"\t"+name+"\t"+designation+"\t"+salary+"\t"+address);
28. }
30. @Override
31. **public** Prototype getClone() {
33. **return** **new** EmployeeRecord(id,name,designation,salary,address);
34. }
35. }//End of EmployeeRecord class.

*File: PrototypeDemo.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
5. **class** PrototypeDemo{
6. **public** **static** **void** main(String[] args) **throws** IOException {
8. BufferedReader br =**new** BufferedReader(**new** InputStreamReader(System.in));
9. System.out.print("Enter Employee Id: ");
10. **int** eid=Integer.parseInt(br.readLine());
11. System.out.print("\n");
13. System.out.print("Enter Employee Name: ");
14. String ename=br.readLine();
15. System.out.print("\n");
17. System.out.print("Enter Employee Designation: ");
18. String edesignation=br.readLine();
19. System.out.print("\n");
21. System.out.print("Enter Employee Address: ");
22. String eaddress=br.readLine();
23. System.out.print("\n");
25. System.out.print("Enter Employee Salary: ");
26. **double** esalary= Double.parseDouble(br.readLine());
27. System.out.print("\n");
29. EmployeeRecord e1=**new** EmployeeRecord(eid,ename,edesignation,esalary,eaddress);
31. e1.showRecord();
32. System.out.println("\n");
33. EmployeeRecord e2=(EmployeeRecord) e1.getClone();
34. e2.showRecord();
35. }
36. }//End of the ProtoypeDemo class.

Structural design pattern.

**Decorator:**

|  |  |
| --- | --- |
| The **decorator design pattern** allows you to add features to an object dynamically.  An example would be the search functionality in an application. You may need to search  for employees such as salary, zip code, skills, and so on.  The user may choose to enter any combination of search criteria, and it would be a daunting task  trying to figure out all the possible combination as the number of fields grow. The decorator pattern  allows you to dynamically add only the fields that the user is searching for. |  |

In the decorator pattern each feature is represented by a class. Therefore if you have 10 features then you will have 10 decorator classes.

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

**In short**

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.

Advantage of Decorator Pattern

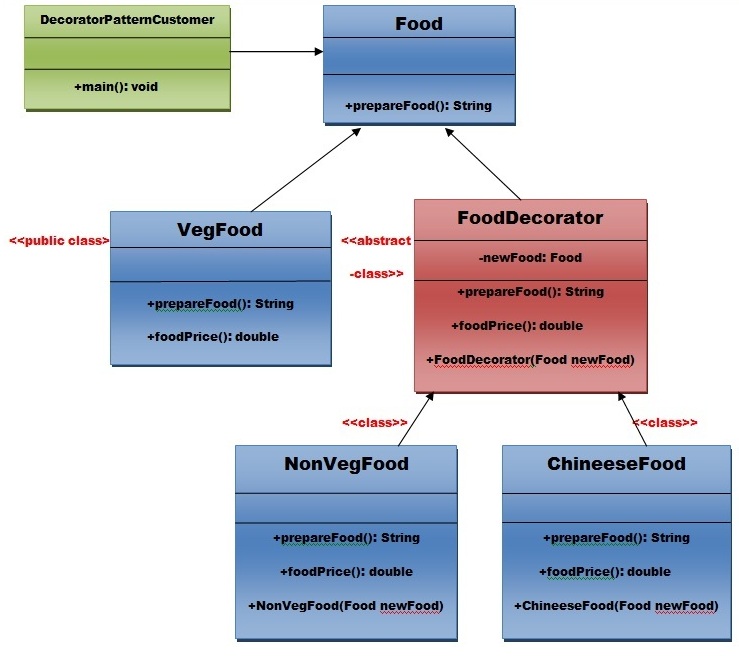
* It provides greater flexibility than static inheritance.
* It enhances the extensibility of the object, because changes are made by coding new classes.
* It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage of Decorator Pattern

It is used:

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

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. }

# **Composite Pattern**

A Composite Pattern says that just **"allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".**

The **composite design pattern** allows you to set up a tree structure and ask each element in the tree structure to perform a task. A typical tree structure would be a company organization chart, where the CEO is at the top and other employees at the bottom. After the tree structure is established you can then ask each element, or employee, to perform a common operation.

Advantage of Composite Design Pattern

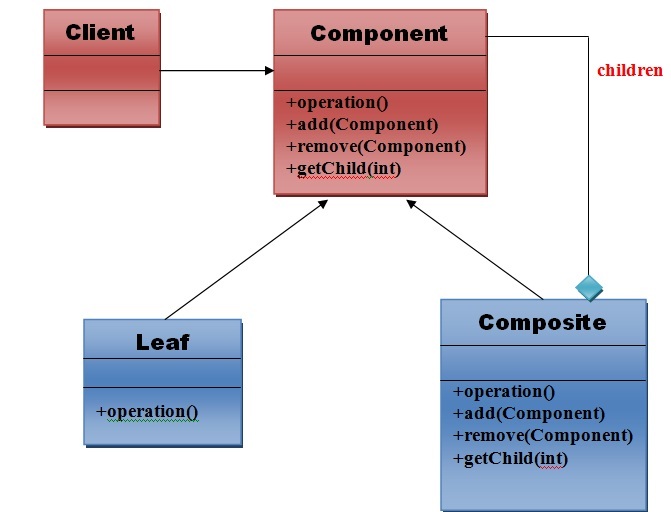
* It defines class hierarchies that contain primitive and complex objects.
* It makes easier to you to add new kinds of components.
* It provides flexibility of structure with manageable class or interface.

#### Usage of Composite Pattern

It is used:

* When you want to represent a full or partial hierarchy of objects.
* When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.

#### UML for Composite Pattern



#### Elements used in Composite Pattern:

Let's see the 4 elements of composte pattern.

##### 1) Component

* Declares interface for objects in composition.
* Implements default behavior for the interface common to all classes as appropriate.
* Declares an interface for accessing and managing its child components.

##### 2) Leaf

* Represents leaf objects in composition. A leaf has no children.
* Defines behavior for primitive objects in the composition.

##### 3) Composite

* Defines behavior for components having children.
* Stores child component.
* Implements child related operations in the component interface.

##### 4) Client

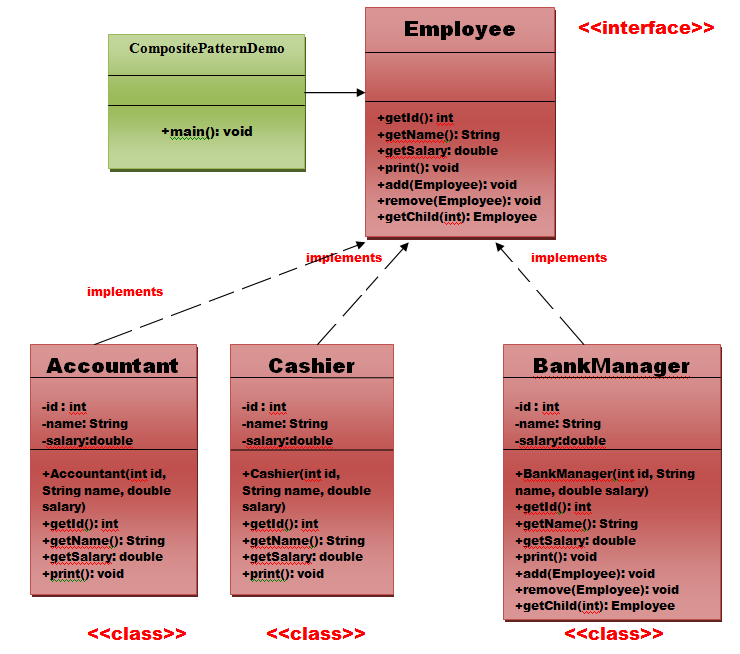
* Manipulates objects in the composition through the component interface.

**Note:**The work flow of above general UML is as follows.

Client uses the component class interface to interact with objects in the composition structure. If recipient is the leaf then request will be handled directly. If recipient is a composite, then it usually forwards the request to its child for performing the additional operations.

### **Example of Composite Pattern**

We can easily understand the example of composite design pattern by the UML diagram given below:



#### Implementation of above UML:

#### Step 1

Create an **Employee** interface that will be treated as a **component.**

1. // this is the Employee interface i.e. Component.
2. **public** **interface** Employee {
3. **public**  **int** getId();
4. **public** String getName();
5. **public** **double** getSalary();
6. **public** **void** print();
7. **public** **void** add(Employee employee);
8. **public** **void** remove(Employee employee);
9. **public** Employee getChild(**int** i);
10. }// End of the Employee interface.

#### Step 2

Create a **BankManager** class that will be treated as a **Composite** and implements Employee interface.

*File: BankManager.java*

1. // this is the BankManager class i.e. Composite.
2. **import** java.util.ArrayList;
3. **import** java.util.Iterator;
4. **import** java.util.List;
5. **public** **class** BankManager **implements** Employee {
6. **private** **int** id;
7. **private** String name;
8. **private** **double** salary;
10. **public** BankManager(**int** id,String name,**double** salary) {
11. **this**.id=id;
12. **this**.name = name;
13. **this**.salary = salary;
14. }
15. List<Employee> employees = **new** ArrayList<Employee>();
16. @Override
17. **public** **void** add(Employee employee) {
18. employees.add(employee);
19. }
20. @Override
21. **public** Employee getChild(**int** i) {
22. **return** employees.get(i);
23. }
24. @Override
25. **public** **void** remove(Employee employee) {
26. employees.remove(employee);
27. }
28. @Override
29. **public** **int** getId()  {
30. **return** id;
31. }
32. @Override
33. **public** String getName() {
34. **return** name;
35. }
36. @Override
37. **public** **double** getSalary() {
38. **return** salary;
39. }
40. @Override
41. **public** **void** print() {
42. System.out.println("==========================");
43. System.out.println("Id ="+getId());
44. System.out.println("Name ="+getName());
45. System.out.println("Salary ="+getSalary());
46. System.out.println("==========================");
48. Iterator<Employee> it = employees.iterator();
50. **while**(it.hasNext())  {
51. Employee employee = it.next();
52. employee.print();
53. }
54. }
55. }// End of the BankManager class.

#### Step 3

Create a **Cashier** class that will be treated as a **leaf** and it will implement to the Employee interface.

*File: Cashier.java*

1. **public**  **class** Cashier **implements** Employee{
2. /\*
3. In this class,there are many methods which are not applicable to cashier because
4. it is a leaf node.
5. \*/
6. **private** **int** id;
7. **private** String name;
8. **private** **double** salary;
9. **public** Cashier(**int** id,String name,**double** salary)  {
10. **this**.id=id;
11. **this**.name = name;
12. **this**.salary = salary;
13. }
14. @Override
15. **public** **void** add(Employee employee) {
16. //this is leaf node so this method is not applicable to this class.
17. }
18. @Override
19. **public** Employee getChild(**int** i) {
20. //this is leaf node so this method is not applicable to this class.
21. **return** **null**;
22. }
23. @Override
24. **public** **int** getId() {
25. // TODO Auto-generated method stub
26. **return** id;
27. }
28. @Override
29. **public** String getName() {
30. **return** name;
31. }
32. @Override
33. **public** **double** getSalary() {
34. **return** salary;
35. }
36. @Override
37. **public** **void** print() {
38. System.out.println("==========================");
39. System.out.println("Id ="+getId());
40. System.out.println("Name ="+getName());
41. System.out.println("Salary ="+getSalary());
42. System.out.println("==========================");
43. }
44. @Override
45. **public** **void** remove(Employee employee) {
46. //this is leaf node so this method is not applicable to this class.
47. }
48. }

#### Step 4

Create a **Accountant** class that will also be treated as a **leaf** and it will implement to the Employee interface.

*File: Accountant.java*

1. **public** **class** Accountant **implements** Employee{
2. /\*
3. In this class,there are many methods which are not applicable to cashier because
4. it is a leaf node.
5. \*/
6. **private** **int** id;
7. **private** String name;
8. **private** **double** salary;
9. **public** Accountant(**int** id,String name,**double** salary)  {
10. **this**.id=id;
11. **this**.name = name;
12. **this**.salary = salary;
13. }
14. @Override
15. **public** **void** add(Employee employee) {
16. //this is leaf node so this method is not applicable to this class.
17. }
18. @Override
19. **public** Employee getChild(**int** i) {
20. //this is leaf node so this method is not applicable to this class.
21. **return** **null**;
22. }
23. @Override
24. **public** **int** getId() {
25. // TODO Auto-generated method stub
26. **return** id;
27. }
28. @Override
29. **public** String getName() {
30. **return** name;
31. }
32. @Override
33. **public** **double** getSalary() {
34. **return** salary;
35. }
36. @Override
37. **public** **void** print() {
38. System.out.println("=========================");
39. System.out.println("Id ="+getId());
40. System.out.println("Name ="+getName());
41. System.out.println("Salary ="+getSalary());
42. System.out.println("=========================");
43. }
44. @Override
45. **public** **void** remove(Employee employee) {
46. //this is leaf node so this method is not applicable to this class.
47. }
48. }

#### Step 5

Create a **CompositePatternDemo** class that will also be treated as a **Client** and ii will use the Employee interface.

*File: CompositePatternDemo.java*

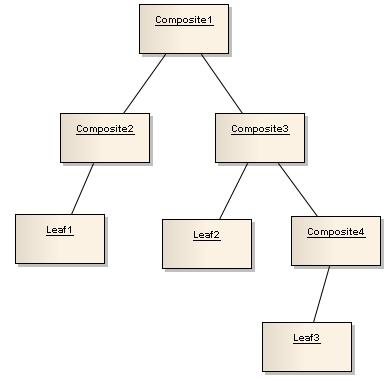
1. **public** **class** CompositePatternDemo {
2. **public** **static** **void** main(String args[]){
3. Employee emp1=**new** Cashier(101,"Sohan Kumar", 20000.0);
4. Employee emp2=**new** Cashier(102,"Mohan Kumar", 25000.0);
5. Employee emp3=**new** Accountant(103,"Seema Mahiwal", 30000.0);
6. Employee manager1=**new** BankManager(100,"Ashwani Rajput",100000.0);
8. manager1.add(emp1);
9. manager1.add(emp2);
10. manager1.add(emp3);
11. manager1.print();
12. }
13. }

[download this composite pattern Example](https://www.javatpoint.com/designpattern/designpatternexample/compositepattern.zip)

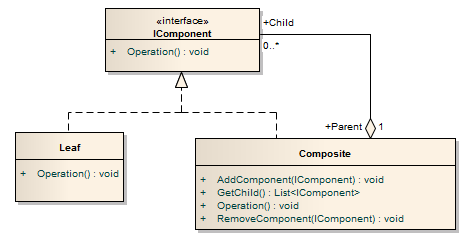
#### Output

1. ==========================
2. Id =100
3. Name =Ashwani Rajput
4. Salary =100000.0
5. ==========================
6. ==========================
7. Id =101
8. Name =Sohan Kumar
9. Salary =20000.0
10. ==========================
11. ==========================
12. Id =102
13. Name =Mohan Kumar
14. Salary =25000.0
15. ==========================
16. =========================
17. Id =103
18. Name =Seema Mahiwal
19. Salary =30000.0
20. =========================

The composite pattern classifies each element in the tree as a composite or a leaf. A composite means that there can be other elements below it, whereas a leaf cannot have any elements below it. Therefore the leaf must be at the very bottom of the tree. The concept is shown in the diagram below:                               
                                                                                                                                                      

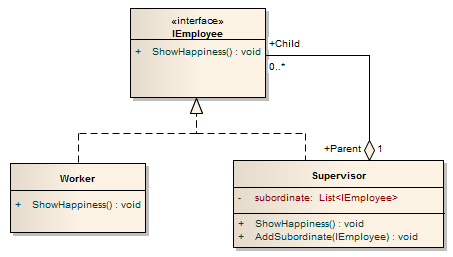
[](http://www.dotnetlead.com/design-patterns/composite/composite8.PNG?attredirects=0)

Let's take a look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference)of the composite pattern first, then we will see an example. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Composite Design Pattern, where you see the distinction between a composite element and the leaf:

[](http://www.dotnetlead.com/design-patterns/composite/composite1.png?attredirects=0)

* The *IComponent*interface defines the methods that both the *Composite*class and the *Leaf*class must implement. The *Operation*method is the common method that all elements in the tree structure can perform. The *IComponent* simply represents an element in the tree.
* The *Leaf*class are elements that cannot have any elements below it, and it only has *Operation*method to perform the task for the element.
* The *Composite*class are elements that can have 0 or more elements below it. The methods that it supports are as follows:
  + The *AddComponent*method adds an element below it
  + The *GetChild*method gets all the elements below it
  + The *Operation*method performs the task for the element itself
  + The *RemoveComponent*method deletes an element below it

Let's see an example. In a company we have supervisors and workers. The supervisors can manage other supervisors or workers under them.   
  
The supervisors will be the composites.   
  
The workers does not manage anyone and they will be the leaves.   
  
All the supervisors and workers are employees, and as an employee you can always show your happiness level in the company (this is the common operation of the elements).   
  
  
The [UML](http://www.dotnetlead.com/UML-Quick-Reference) for this example is shown below:

[](http://www.dotnetlead.com/design-patterns/composite/composite2.png?attredirects=0)

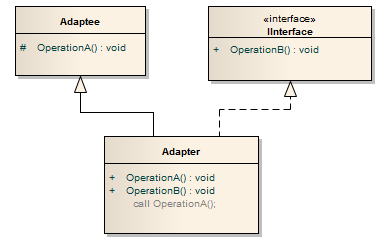
* The *IEmployee*interface defines the operation that all employees must be able to perform, which is the *ShowHappiness*method.
* The *Worker*class are the employees that does not manage anyone, and implements only the *ShowHappiness*method.
* The *Supervisor*class are the employees that can manage other employees and have the following variables and methods:
  + The private variable *subordinate* are the list of employees that the supervisor manages.
  + The *AddSubordinate*method adds an employee under the supervisor
  + The *ShowHappiness*method shows the supervisor's happiness level.

When you call a supervisor's *ShowHappiness*method, it will show both the supervisor’s happiness and all of its subordinate’s happiness by calling each of the subordinate's *ShowHappiness*method.

The key to the composite design pattern is that it allows you to set up a structure with a common operation (such as the *ShowHappiness*method), and then you can have all the elements to perform the common operation. This is done by keeping a list of child elements that implements the common interface in the composite class, and then calling each child element's operations.

|  |  |
| --- | --- |
| **Adapter design pattern** The **Adapter Design Pattern** allows you to make an existing class work with other existing class libraries without changing the code of the existing class. |  |

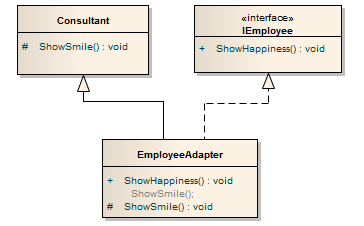
We often need to use the methods of an existing class to work with other existing libraries. The way to do this is by creating another class, named the Adapter, that inherits from the existing class while implementing the interface of the existing library. The end result is that the Adapter can call the method of the existing class (since the Adapter inherits from the existing class) and can work in the existing library (since the Adapter implements the interface of the existing library).  
                                                                                                                                                                        
Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Adapter Design Pattern:

[](http://www.dotnetlead.com/design-patterns/adapter/adapter1.png?attredirects=0)

* The *Adaptee*is the existing class.
* The *IInterface*is the interface defined in the existing library.
* The *Adapter*is the class that you create, it is inherited from the *Adaptee*class and it implements the *IInterface*interface. Notice that it can call the *OperationA*method(inherited from the *Adaptee*) inside its *OperationB*method(implemented by the *IInterface*).

Let's see an example. Using the example given in the [composite design pattern](http://www.dotnetlead.com/design-patterns/composite), where we had an organization tree that was constructed where all the employees implements the *IEmployee*interface. The *IEmployee*interface is from the existing library.  
  
You are then given the *Consultant*class and you need to plug this *Consultant*class into the organization tree. The *Consultant*class is the Adaptee.

The way to do this is by creating the adapter class named the *EmployeeAdapter*, that inherits from the *Consultant*class while it implements the *IEmployee*interface:

[](http://www.dotnetlead.com/design-patterns/adapter/adapter2.png?attredirects=0)

In the Adapter we can then call the methods from the parent class to mimic the behaviors needed in the common interface. In the *EmployeeAdapter*class we can call the *ShowSmile*method from the parent *Consultant* class in its implementation of the *IEmployee*interface, which requires the *ShowHappiness* method.

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper.**

Advantage of Adapter Pattern

* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality.

Usage of Adapter pattern:

It is used:

* When an object needs to utilize an existing class with an incompatible interface.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

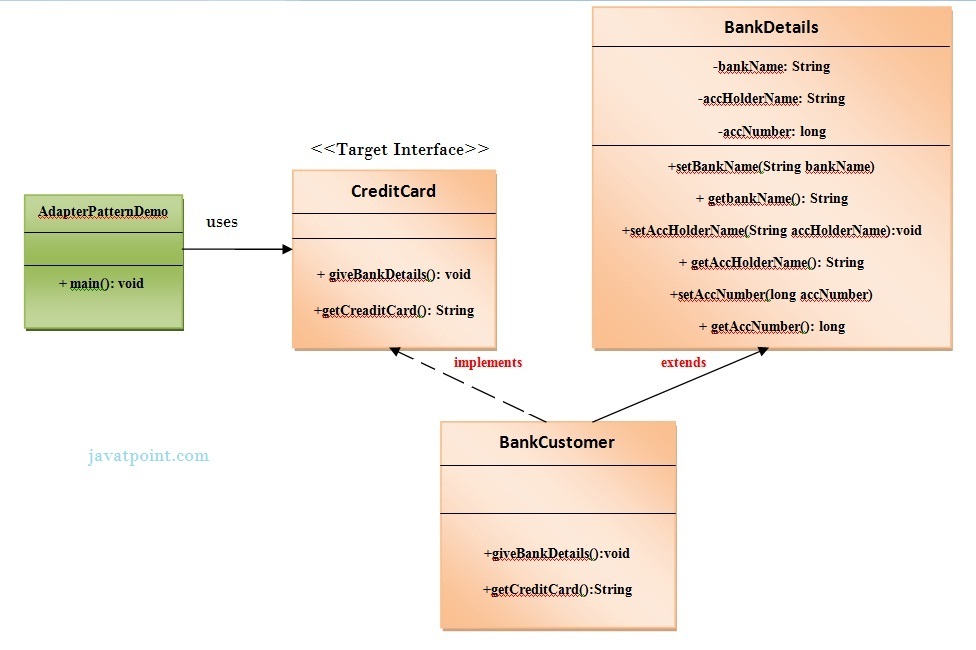
### **Example of Adapter Pattern**

Let's understand the example of adapter design pattern by the above UML diagram.

#### UML for Adapter Pattern:

There are the following specifications for the adapter pattern:

* **Target Interface:** This is the desired interface class which will be used by the clients.
* **Adapter class:** This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.
* **Adaptee class:** This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.
* **Client:** This class will interact with the Adapter class.



#### Implementation of above UML:

#### Step 1

Create a **CreditCard** interface (Target interface).

1. **public** **interface** CreditCard {
2. **public** **void** giveBankDetails();
3. **public** String getCreditCard();
4. }// End of the CreditCard interface.

#### Step 2

Create a **BankDetails** class (Adaptee class).

*File: BankDetails.java*

1. // This is the adapter class.
2. **public** **class** BankDetails{
3. **private** String bankName;
4. **private** String accHolderName;
5. **private** **long** accNumber;
7. **public** String getBankName() {
8. **return** bankName;
9. }
10. **public** **void** setBankName(String bankName) {
11. **this**.bankName = bankName;
12. }
13. **public** String getAccHolderName() {
14. **return** accHolderName;
15. }
16. **public** **void** setAccHolderName(String accHolderName) {
17. **this**.accHolderName = accHolderName;
18. }
19. **public** **long** getAccNumber() {
20. **return** accNumber;
21. }
22. **public** **void** setAccNumber(**long** accNumber) {
23. **this**.accNumber = accNumber;
24. }
25. }// End of the BankDetails class.

#### Step 3

Create a **BankCustomer** class (Adapter class).

*File: BankCustomer.java*

1. // This is the adapter class
3. **import** java.io.BufferedReader;
4. **import** java.io.InputStreamReader;
5. **public** **class** BankCustomer **extends** BankDetails **implements** CreditCard {
6. **public** **void** giveBankDetails(){
7. **try**{
8. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
10. System.out.print("Enter the account holder name :");
11. String customername=br.readLine();
12. System.out.print("\n");
14. System.out.print("Enter the account number:");
15. **long** accno=Long.parseLong(br.readLine());
16. System.out.print("\n");
18. System.out.print("Enter the bank name :");
19. String bankname=br.readLine();
21. setAccHolderName(customername);
22. setAccNumber(accno);
23. setBankName(bankname);
24. }**catch**(Exception e){
25. e.printStackTrace();
26. }
27. }
28. @Override
29. **public** String getCreditCard() {
30. **long** accno=getAccNumber();
31. String accholdername=getAccHolderName();
32. String bname=getBankName();
34. **return** ("The Account number "+accno+" of "+accholdername+" in "+bname+ "
35. bank is valid and authenticated **for** issuing the credit card. ");
36. }
37. }//End of the BankCustomer class.

#### Step 4

Create a **AdapterPatternDemo** class (client class).

*File: AdapterPatternDemo.java*

1. //This is the client class.
2. **public** **class** AdapterPatternDemo {
3. **public** **static** **void** main(String args[]){
4. CreditCard targetInterface=**new** BankCustomer();
5. targetInterface.giveBankDetails();
6. System.out.print(targetInterface.getCreditCard());
7. }
8. }//End of the BankCustomer class.

[download this example](https://www.javatpoint.com/designpattern/designpatternexample/adapterpattern.zip)

#### Output

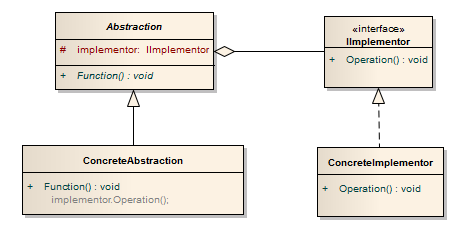
1. Enter the account holder name :Sonoo Jaiswal
3. Enter the account number:10001
5. Enter the bank name :State Bank of India
7. The Account number 10001 of Sonoo Jaiswal in State Bank of India bank is valid
8. and authenticated **for** issuing the credit card.

**Bridge design pattern**

|  |  |
| --- | --- |
| The **bridge design pattern** allows you to separate the abstraction from the implementation. In the bridge pattern there are 2 parts, the first part is the Abstraction, and the second part is the Implementation.  The bridge pattern allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the Implementation part. |  |

Let's look at an example to see the concept behind the bridge pattern. For example, inside a house there are appliances that you can turn on or off, such as the floor lamp, the TV, and the vacuum cleaner. There are different ways to turn the appliance on or off, such as using the on/off switch, the pull switch, or using a remote control. The concept of turning the appliance on or off is the Abstraction part in the bridge pattern, and the user only needs to know the Abstraction part. This is the first part of the bridge pattern.

The second part in the bridge design pattern, the Implementation, is the part that turns the appliance on or off. When the floor lamp,the TV, or the vacuum cleaner receives the signal to turn on or off, it uses its internal implementation to perform the action.  The user should not be bothered about how the appliance is turned on or off, and this is the key of the bridge design pattern.   
  
  
  
  
  
  
Let's look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the bridge design pattern first, then we will see an example to see how it works. Below is the[UML](http://www.dotnetlead.com/UML-Quick-Reference) of the bridge pattern, the left side is the Abstraction, and the right side is the Implementation:

[](http://www.dotnetlead.com/design-patterns/bridge/bridge1.png?attredirects=0)

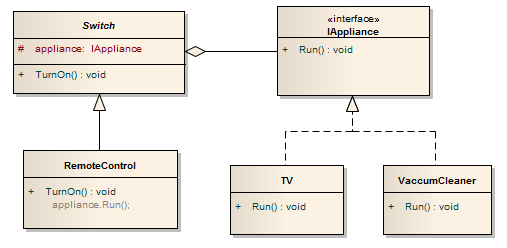
The left side is the Abstraction part with the following classes:

* *Abstraction*is the abstract parent class of the *ConcreteAbstraction*class. It defines the *Function*method for the client code to call. The protected *implementor*variable holds the reference to the object that performs the implementation
* *ConcreteAbstraction*is the concrete class that is inherited from the *Abstraction*class

The right side is the Implementation part with the following classes:

* *IImplementor*is the interface that all the implementation classes must implement
* *ConcreteImplementor*is the concrete class that performs the implementation

Applying this [UML](http://www.dotnetlead.com/UML-Quick-Reference) to our example means that the left side represents controls that can turn appliances on or off, such as the on/off switch, the pull switch, or the remote control, while the right side represents the actual appliances that performs the action, such as the TV or the VacuumCleaner.   
  
  
  
  
  
  
  
  
Therefore the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of our example will be:

[](http://www.dotnetlead.com/design-patterns/bridge/bridge2.png?attredirects=0)

Another example of the bridge pattern is the copy and paste function in many applications.  The copy and paste function is the abstraction, where the user only needs to know how to use it, and the actual action of transferring the information into the memory and transferring the information onto the application is the implementation.  
  
The key benefit of the bridge design pattern is it allows you to develop the Abstraction and the Implementation parts independently. It also cuts down on the number of classes that you need to create to fulfill all the possible combination of the Abstractions (user interface concepts) and the Implementations (actual actions behind the scene).

A Bridge Pattern says that just **"decouple the functional abstraction from the implementation so that the two can vary independently".**

The Bridge Pattern is also known as **Handle or Body.**

#### Advantage of Bridge Pattern

* It enables the separation of implementation from the interface.
* It improves the extensibility.
* It allows the hiding of implementation details from the client.

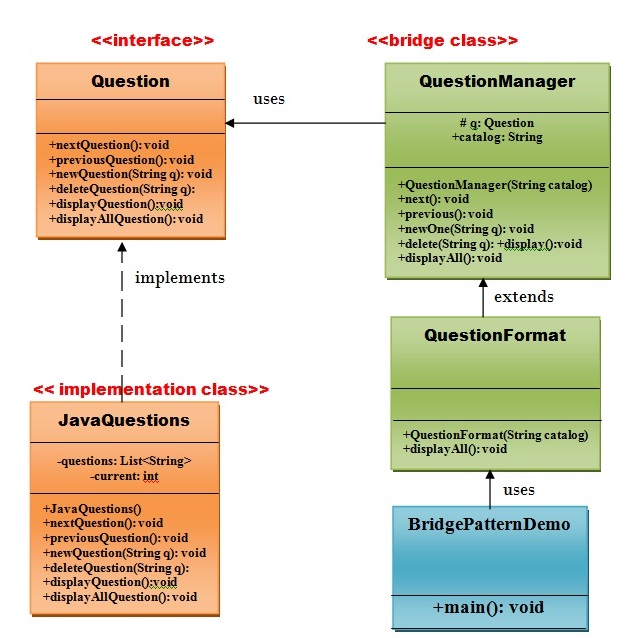
#### Usage of Bridge Pattern

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes are made in the implementation does not affect the clients.

### **Example of Bridge Pattern**

The UML given below describes the example of bridge pattern.

#### UML for Bridge Pattern:



#### Implementation of above UML:

#### Step 1

Create a **Question** interface that provides the navigation from one question to another or vice-versa.

1. // this is the Question interface.
2. **public** **interface** Question {
3. **public** **void** nextQuestion();
4. **public** **void** previousQuestion();
5. **public** **void** newQuestion(String q);
6. **public** **void** deleteQuestion(String q);
7. **public** **void** displayQuestion();
8. **public** **void** displayAllQuestions();
9. }
10. // End of the Question interface.

#### Step 2

Create a **JavaQuestions** implementation class that will implement **Question** interface.

1. // this is the JavaQuestions class.
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **public** **class** JavaQuestions **implements** Question {
5. **private** List <String> questions = **new** ArrayList<String>();
6. **private** **int** current = 0;
7. **public** JavaQuestions(){
8. questions.add("What is class? ");
9. questions.add("What is interface? ");
10. questions.add("What is abstraction? ");
11. questions.add("How multiple polymorphism is achieved in java? ");
12. questions.add("How many types of exception  handling are there in java? ");
13. questions.add("Define the keyword final for  variable, method, and class in java? ");
14. questions.add("What is abstract class? ");
15. questions.add("What is multi-threading? ");
16. }
17. **public** **void** nextQuestion() {
18. **if**( current <= questions.size()-1 )
19. current++;
20. System.out.print(current);
21. }
23. **public** **void** previousQuestion() {
24. **if**( current > 0 )
25. current--;
26. }
28. **public** **void** newQuestion(String quest) {
29. questions.add(quest);
30. }
32. **public** **void** deleteQuestion(String quest) {
33. questions.remove(quest);
34. }
36. **public** **void** displayQuestion() {
37. System.out.println( questions.get(current) );
38. }
39. **public** **void** displayAllQuestions() {
40. **for** (String quest : questions) {
41. System.out.println(quest);
42. }
43. }
44. }// End of the JavaQuestions class.

#### Step 3

Create a **QuestionManager** class that will use **Question** interface which will act as a bridge..

1. // this is the QuestionManager class.
2. **public** **class** QuestionManager  {
3. **protected** Question q;
4. **public** String catalog;
5. **public** QuestionManager(String catalog) {
6. **this**.catalog=catalog;
7. }
8. **public** **void** next() {
9. q.nextQuestion();
10. }
11. **public** **void** previous() {
12. q.previousQuestion();
13. }
14. **public** **void** newOne(String quest) {
15. q.newQuestion(quest);
16. }
17. **public** **void** delete(String quest) {
18. q.deleteQuestion(quest);
19. }
20. **public** **void** display() {
21. q.displayQuestion();
22. }
23. **public** **void** displayAll() {
24. System.out.println("Question Paper: " + catalog);
25. q.displayAllQuestions();
26. }
27. }// End of the QuestionManager class.

#### Step 4

Create a **QuestionFormat** class that will extend the **QuestionManager** class

1. // this is the QuestionFormat class.
2. **public** **class** QuestionFormat **extends** QuestionManager {
3. **public** QuestionFormat(String catalog){
4. **super**(catalog);
5. }
6. **public** **void** displayAll() {
7. System.out.println("\n---------------------------------------------------------");
8. **super**.displayAll();
9. System.out.println("-----------------------------------------------------------");
10. }
11. }// End of the QuestionFormat class.

#### Step 5

Create a **BridgePatternDemo** class.

1. // this is the BridgePatternDemo class.
2. **public** **class** BridgePatternDemo {
3. **public** **static** **void** main(String[] args) {
4. QuestionFormat questions = **new** QuestionFormat("Java Programming Language");
5. questions.q = **new** JavaQuestions();
6. questions.delete("what is class?");
7. questions.display();
8. questions.newOne("What is inheritance? ");
10. questions.newOne("How many types of inheritance are there in java?");
11. questions.displayAll();
12. }
13. }// End of the BridgePatternDemo class.

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

#### Output

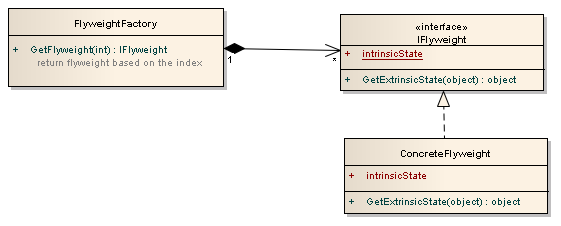
1. What is **interface**?
3. --------------------------------------------------------------------
4. Question Paper: Java Programming Language
5. What is **class**?
6. What is **interface**?
7. What is abstraction?
8. How multiple polymorphism is achieved in java?
9. How many types of exception  handling are there in java?
10. Define the keyword **final** **for**  variable, method, and **class** in java?
11. What is **abstract** **class**?
12. What is multi-threading?
13. What is inheritance?
14. How many types of inheritance are there in java?
15. -----------------------------------------------------------------------

**Flyweight design pattern:**

|  |  |
| --- | --- |
| The **flyweight design pattern** allows you to reuse memory spaces in an application when you have lots of objects that are almost identical in nature. For example, if you are writing a game for a smartphone where the amount of memory is very limited and you need to show many aliens that are identical in shape, you can have only one place that holds the shape of the alien instead of keeping each identical shape in the precious memory. |  |

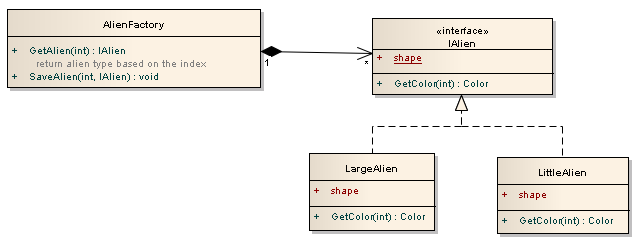
In the flyweight pattern there is the concept of Intrinsic and Extrinsic state. Intrinsic states are things that are constant and are stored in the memory. Extrinsic states are things that are not constant and needs to be calculated on the fly, and are therefore not stored in the memory.

For example, in the game that we would like to create, the shapes of the aliens are all the same, but their color will change based on how mad each are. The shapes of the aliens will be Intrinsic, and the color of the alien will be Extrinsic.   
  
  
  
  
  
  
  
  
  
Let's look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the flyweight pattern first, then we will look at an example to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) for the flyweight design pattern:

[](http://www.dotnetlead.com/design-patterns/flyweight/flyweight1.PNG?attredirects=0)

* The *FlyweightFactory*class is just an indexer that allows you to retrieve the flyweight object when given an index number, since you may have many flyweight objects in your application.
* The *IFlyweight*interface defines the methods and the properties required for the flyweight objects. The *GetExtrinsicState*method gets the extrinsic states and the ­*intrinsicState* variable holds the intrinsic states.
* The *ConcreteFlyweight*class is the actual flyweight object. The *intrinsicState* variable stores the information that are constant, while the *GetExtrinsicState*method calculates the extrinsic states on the fly.

Applying the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the flyweight pattern to our example of creating aliens of different shapes and colors gives us the following [UML](http://www.dotnetlead.com/UML-Quick-Reference):

[](http://www.dotnetlead.com/design-patterns/flyweight/flyweight2.PNG?attredirects=0)

* The *AlienFactory*class stores and retrieves different types of aliens using its *GetAlien*and *SaveAlien*methods.
* The *IAlien*interface defines the *shape* property as the intrinsic state and the *GetColor* method as the extrinsic state.
* The *LargeAlien*and the *LittleAlien*classes are the flyweight objects where each has its own *shape*intrinsic state and ways of calculating the *GetColor*extrinsic state.

A comparison between the flyweight pattern and the [prototype pattern](http://www.dotnetlead.com/design-patterns/prototype) shows some similarities in the [UML](http://www.dotnetlead.com/UML-Quick-Reference), in that both uses a manager to store and retrieve the objects in the collection. But there is a clear difference between the two. The [prototype pattern](http://www.dotnetlead.com/design-patterns/prototype) is used to create new objects that are similar in nature (hence it's a creational pattern), while the flyweight patternis used to allow the application to point to the same instance of the object to save memory (hence it's a structural pattern).

A Flyweight Pattern says that just **"to reuse already existing similar kind of objects by storing them and create new object when no matching object is found"**.

Advantage of Flyweight Pattern

* It reduces the number of objects.
* It reduces the amount of memory and storage devices required if the objects are persisted

Usage of Flyweight Pattern

* When an application uses number of objects
* When the storage cost is high because of the quantity of objects.
* When the application does not depend on object identity.

**Proxy Design pattern🡪**

|  |  |  |
| --- | --- | --- |
|  | The **proxy design pattern** allows you to provide an interface to other objects by creating a wrapper class as the proxy. The wrapper class, which is the proxy, can add additional functionality to the object of interest without changing the object's code. |  |

Below are some of the common examples in which the proxy pattern are used:

* Adding security access to an existing object. The proxy will determine if the client can access the object of interest.
* Simplifying the API of complex objects. The proxy can provide a simple API so that the client code does not have to deal with the complexity of the object of interest.
* Providing interface for remote resources such as web service or REST resources.
* Coordinating expensive operations on remote resources by asking the remote resources to start the operation as soon as possible before accessing the resources.
* Adding a thread-safe feature to an existing class without changing the existing class's code.

In short, the proxy is the object that is being called by the client to access the real object behind the scene.

Simply, proxy means an object representing another object.

According to GoF, a Proxy Pattern **"provides the control for accessing the original object".**

So, we can perform many operations like hiding the information of original object, on demand loading etc.

Proxy pattern is also known as **Surrogate or Placeholder.**

#### RMI API uses proxy design pattern. Stub and Skeleton are two proxy objects used in RMI.

#### Advantage of Proxy Pattern

* It provides the protection to the original object from the outside world.

#### Usage of Proxy Pattern:

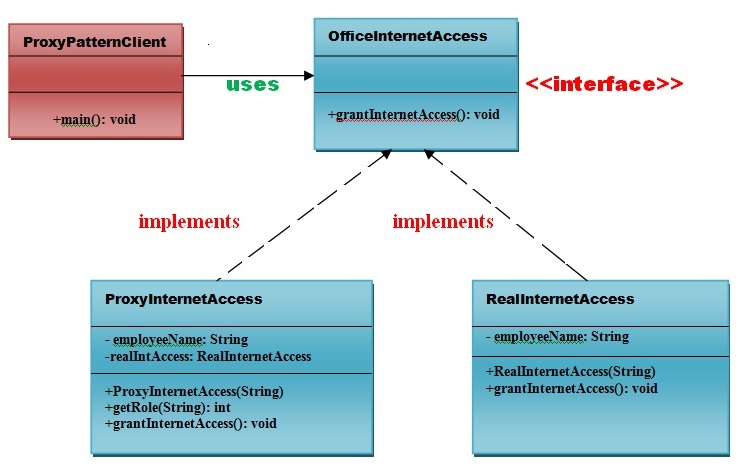
It is used:

* It can be used in **Virtual Proxy** scenario---Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.
* It can be used in **Protective Proxy** scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
* It can be used in **Remote Proxy** scenario---A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.
* It can be used in **Smart Proxy** scenario---A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

### Example of Proxy Pattern

Let's understand the example of proxy design pattern by the above UML diagram.

#### UML for Proxy Pattern:



#### Implementation of above UML:

#### Step 1

Create an **OfficeInternetAccess** interface.

1. public interface OfficeInternetAccess {
2. public void grantInternetAccess();
3. }

#### Step 2

Create a **RealInternetAccess** class that will implement **OfficeInternetAccess** interface for granting the permission to the specific employee.

File: RealInternetAccess.java

1. public class RealInternetAccess implements OfficeInternetAccess {
2. private String employeeName;
3. public RealInternetAccess(String empName) {
4. this.employeeName = empName;
5. }
6. @Override
7. public void grantInternetAccess() {
8. System.out.println("Internet Access granted for employee: "+ employeeName);
9. }
10. }

#### Step 3

Create a **ProxyInternetAccess** class that will implement **OfficeInternetAccess** interface for providing the object of **RealInternetAccess** class.

File: ProxyInternetAccess.java

1. public class ProxyInternetAccess implements OfficeInternetAccess {
2. private String employeeName;
3. private RealInternetAccess  realaccess;
4. public ProxyInternetAccess(String employeeName) {
5. this.employeeName = employeeName;
6. }
7. @Override
8. public void grantInternetAccess()
9. {
10. if (getRole(employeeName) > 4)
11. {
12. realaccess = new RealInternetAccess(employeeName);
13. realaccess.grantInternetAccess();
14. }
15. else
16. {
17. System.out.println("No Internet access granted. Your job level is below 5");
18. }
19. }
20. public int getRole(String emplName) {
21. // Check role from the database based on Name and designation
22. // return job level or job designation.
23. return 9;
24. }
25. }

#### Step 4

Now, Create a **ProxyPatternClient** class that can access the internet actually.

File: ProxyPatternClient.java

1. public class ProxyPatternClient {
2. public static void main(String[] args)
3. {
4. OfficeInternetAccess access = new ProxyInternetAccess("Ashwani Rajput");
5. access.grantInternetAccess();
6. }
7. }

#### Output

1. No Internet access granted. Your job level is below 5

**Façade design pattern🡪**

|  |  |  |
| --- | --- | --- |
|  | The **façade design pattern** allows you to provide a simplified interface from multiple class libraries. It provides a simple interface that hides the complexity of the class libraries being used. |  |

For example, if the client code needs to access 3 different class libraries for a single functionality, instead of having the client code accessing those 3 class libraries, you can just create another class that calls the 3 class libraries and have the client access only the class that you have created. The result is a simplified interface for the client and the client will not need to know the details of the different class libraries.

A Facade Pattern says that just **"just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".**

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Practically, **every Abstract Factory** is a type of **Facade.**

#### Advantage of Facade Pattern

* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

#### Usage of Facade Pattern:

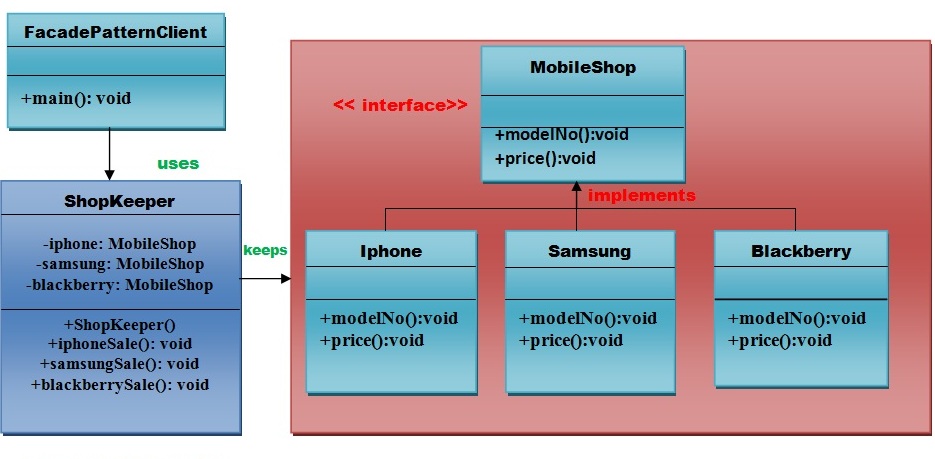
It is used:

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.

### Example of Facade Pattern

Let's understand the example of facade design pattern by the above UML diagram.

#### UML for Facade Pattern:



#### Implementation of above UML:

#### Step 1

Create a **MobileShop** interface.

File: MobileShop.java

1. public interface MobileShop {
2. public void modelNo();
3. public void price();
4. }

#### Step 2

Create a **Iphone** implementation class that will implement **Mobileshop** interface.

File: Iphone.java

1. public class Iphone implements MobileShop {
2. @Override
3. public void modelNo() {
4. System.out.println(" Iphone 6 ");
5. }
6. @Override
7. public void price() {
8. System.out.println(" Rs 65000.00 ");
9. }
10. }

#### Step 3

Create a **Samsung** implementation class that will implement **Mobileshop** interface.

File: Samsung.java

1. public class Samsung implements MobileShop {
2. @Override
3. public void modelNo() {
4. System.out.println(" Samsung galaxy tab 3 ");
5. }
6. @Override
7. public void price() {
8. System.out.println(" Rs 45000.00 ");
9. }
10. }

#### Step 4

Create a **Blackberry** implementation class that will implement **Mobileshop** interface .

File: Blackberry.java

1. public class Blackberry implements MobileShop {
2. @Override
3. public void modelNo() {
4. System.out.println(" Blackberry Z10 ");
5. }
6. @Override
7. public void price() {
8. System.out.println(" Rs 55000.00 ");
9. }
10. }

#### Step 5

Create a **ShopKeeper** concrete class that will use **MobileShop** interface.

File: ShopKeeper.java

1. public class ShopKeeper {
2. private MobileShop iphone;
3. private MobileShop samsung;
4. private MobileShop blackberry;
6. public ShopKeeper(){
7. iphone= new Iphone();
8. samsung=new Samsung();
9. blackberry=new Blackberry();
10. }
11. public void iphoneSale(){
12. iphone.modelNo();
13. iphone.price();
14. }
15. public void samsungSale(){
16. samsung.modelNo();
17. samsung.price();
18. }
19. public void blackberrySale(){
20. blackberry.modelNo();
21. blackberry.price();
22. }
23. }

#### Step 6

Now, Creating a **client** that can purchase the mobiles from **MobileShop** through **ShopKeeper.**

File: FacadePatternClient.java

1. import java.io.BufferedReader;
2. import java.io.IOException;
3. import java.io.InputStreamReader;
5. public class FacadePatternClient {
6. private static int  choice;
7. public static void main(String args[]) throws NumberFormatException, IOException{
8. do{
9. System.out.print("========= Mobile Shop ============ \n");
10. System.out.print("            1. IPHONE.              \n");
11. System.out.print("            2. SAMSUNG.              \n");
12. System.out.print("            3. BLACKBERRY.            \n");
13. System.out.print("            4. Exit.                     \n");
14. System.out.print("Enter your choice: ");
16. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
17. choice=Integer.parseInt(br.readLine());
18. ShopKeeper sk=new ShopKeeper();
20. switch (choice) {
21. case 1:
22. {
23. sk.iphoneSale();
24. }
25. break;
26. case 2:
27. {
28. sk.samsungSale();
29. }
30. break;
31. case 3:
32. {
33. sk.blackberrySale();
34. }
35. break;
36. default:
37. {
38. System.out.println("Nothing You purchased");
39. }
40. return;
41. }
43. }while(choice!=4);
44. }
45. }

[download this example](https://www.javatpoint.com/designpattern/designpatternexample/facadepattern.zip)

#### Output

1. ========= Mobile Shop ============
2. 1. IPHONE.
3. 2. SAMSUNG.
4. 3. BLACKBERRY.
5. 4. Exit.
6. Enter your choice: 1
7. Iphone 6
8. Rs 65000.00
9. ========= Mobile Shop ============
10. 1. IPHONE.
11. 2. SAMSUNG.
12. 3. BLACKBERRY.
13. 4. Exit.
14. Enter your choice: 2
15. Samsung galaxy tab 3
16. Rs 45000.00
17. ========= Mobile Shop ============
18. 1. IPHONE.
19. 2. SAMSUNG.
20. 3. BLACKBERRY.
21. 4. Exit.
22. Enter your choice: 3
23. Blackberry Z10
24. Rs 55000.00
25. ========= Mobile Shop ============
26. 1. IPHONE.
27. 2. SAMSUNG.
28. 3. BLACKBERRY.
29. 4. Exit.
30. Enter your choice: 4
31. Nothing You purchased

**Behavioral design pattern 🡪**

**Chain of responsibility:**

|  |  |  |
| --- | --- | --- |
|  | The **chain of responsibility pattern** allows you to pass a request to from an object to the next until the request is fulfilled.  For example, you can pass a mortgage application request to a bank manager, and if the manager cannot approve the loan, it can be passed to his supervisor and so on. The chain of responsibility automates the passing of requests from one object to the next. |  |

The logic of the pattern is analogous to the exception handling routine in the compiler. When an exception happens, it is passed to the next catch statement until it is matched to an exception type where it will be handled.

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process.

In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

#### Advantage of Chain of Responsibility Pattern

* It reduces the coupling.
* It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

#### Usage of Chain of Responsibility Pattern:

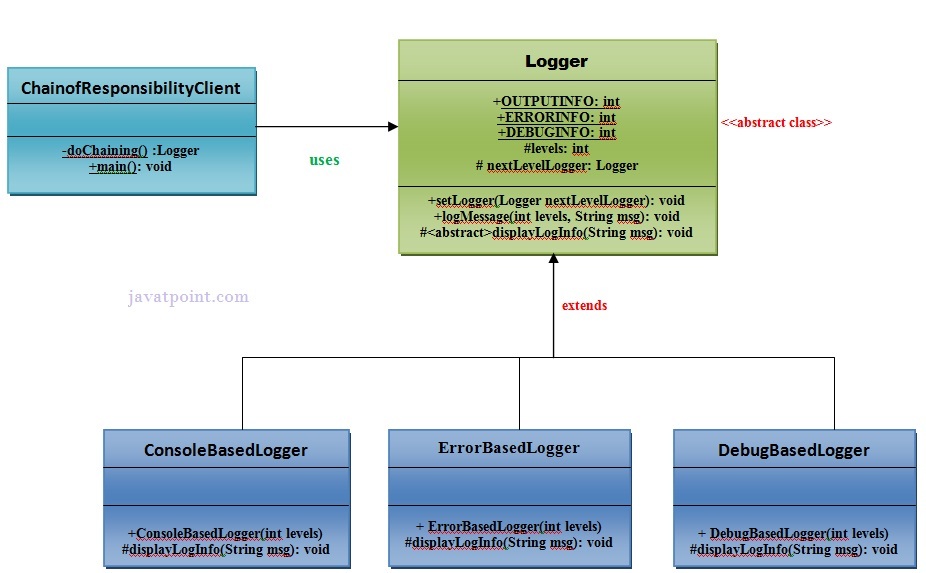
It is used:

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way.

### Example of Chain of Responsibility Pattern

Let's understand the example of Chain of Responsibility Pattern by the above UML diagram.

#### UML for Chain of Responsibility Pattern:



#### Implementation of above UML:

#### Step 1

Create a **Logger** abstract class.

1. public abstract class Logger {
2. public static int OUTPUTINFO=1;
3. public static int ERRORINFO=2;
4. public static int DEBUGINFO=3;
5. protected int levels;
6. protected Logger nextLevelLogger;
7. public void setNextLevelLogger(Logger nextLevelLogger) {
8. this.nextLevelLogger = nextLevelLogger;
9. }
10. public void logMessage(int levels, String msg){
11. if(this.levels<=levels){
12. displayLogInfo(msg);
13. }
14. if (nextLevelLogger!=null) {
15. nextLevelLogger.logMessage(levels, msg);
16. }
17. }
18. protected abstract void displayLogInfo(String msg);
19. }

#### Step 2

Create a **ConsoleBasedLogger** class.

File: ConsoleBasedLogger.java

1. public class ConsoleBasedLogger extends Logger {
2. public ConsoleBasedLogger(int levels) {
3. this.levels=levels;
4. }
5. @Override
6. protected void displayLogInfo(String msg) {
7. System.out.println("CONSOLE LOGGER INFO: "+msg);
8. }
9. }

#### Step 3

Create a **DebugBasedLogger** class.

File: DebugBasedLogger.java

1. public class DebugBasedLogger extends Logger {
2. public DebugBasedLogger(int levels) {
3. this.levels=levels;
4. }
5. @Override
6. protected void displayLogInfo(String msg) {
7. System.out.println("DEBUG LOGGER INFO: "+msg);
8. }
9. }// End of the DebugBasedLogger class.

#### Step 4

Create a **ErrorBasedLogger** class.

File: ErrorBasedLogger.java

1. public class ErrorBasedLogger extends Logger {
2. public ErrorBasedLogger(int levels) {
3. this.levels=levels;
4. }
5. @Override
6. protected void displayLogInfo(String msg) {
7. System.out.println("ERROR LOGGER INFO: "+msg);
8. }
9. }// End of the ErrorBasedLogger class.

#### Step 5

Create a **ChainOfResponsibilityClient** class.

File: ChainofResponsibilityClient.java

1. public class ChainofResponsibilityClient {
2. private static Logger doChaining(){
3. Logger consoleLogger = new ConsoleBasedLogger(Logger.OUTPUTINFO);
5. Logger errorLogger = new ErrorBasedLogger(Logger.ERRORINFO);
6. consoleLogger.setNextLevelLogger(errorLogger);
8. Logger debugLogger = new DebugBasedLogger(Logger.DEBUGINFO);
9. errorLogger.setNextLevelLogger(debugLogger);
11. return consoleLogger;
12. }
13. public static void main(String args[]){
14. Logger chainLogger= doChaining();
16. chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");
17. chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");
18. chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is compeled");
19. }
20. }

#### Output

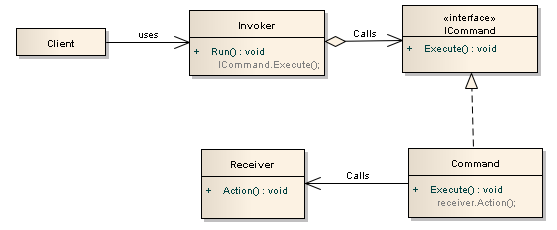
1. bilityClient
2. CONSOLE LOGGER INFO: Enter the sequence of values
3. CONSOLE LOGGER INFO: An error is occured now
4. ERROR LOGGER INFO: An error is occured now
5. CONSOLE LOGGER INFO: This was the error now debugging is compeled
6. ERROR LOGGER INFO: This was the error now debugging is compeled
7. DEBUG LOGGER INFO: This was the error now debugging is compeled

**Command**

|  |  |  |
| --- | --- | --- |
|  | The **command design pattern** allows you to store a list of actions that you can execute later. A common example is storing the undo actions in an application. The undo actions are stored as the user is making changes in an application. When the user decides to perform the undo, the undo actions are retrieved and executed.  The benefit of the command pattern is that it hides the details of the actions that needs to be performed, so that the client code does not need to be concerned about the details when it needs to execute the actions. The client code just need to tell the application to execute the command that was stored. |  |

In an undo example, if the user moved a rectangle from position X to position Y, then the undo action will be to move the rectangle from position Y back to position X. The details of this undo action is stored in the command objects so that when the user needs to execute the undo action, the application only need to tell the command object to execute its action without knowing that it is suppose to move the rectangle from position Y back to position X.                     
  
The command pattern is not focused so much on the sequence of the actions stored, but is more on hiding the details of the action it needs to perform.

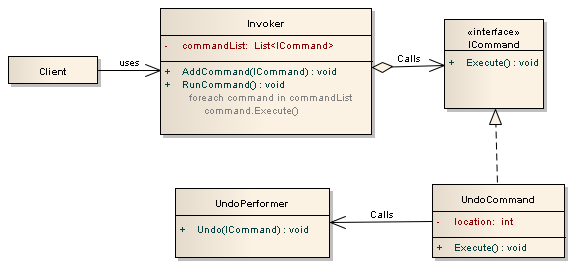
We will see an example of the Command Design Pattern to show you how it works. But for now let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the command design pattern:

[](http://www.dotnetlead.com/design-patterns/command/command1.PNG?attredirects=0)

* The *ICommand* interface defines the methods that all *Command* classes must implement.
* The *Command* class stores the details of the actions that need to be performed (the location of position Y and position X for the undo).
* The *Receiver* class performs the action when called upon (moving the rectangle from position Y back to position X).
* The *Invoker* class stores the list of commands and can ask the *ICommand* to execute.
* The *Client* class uses the *Invoker* to run the commands.

The client code (calling code) will used the Invoker to run the commands, where the *Command* objects will call the *Receiver* to perform the action. The benefit is that the client code does not need to know what is stored in the *Command* objects nor the actions that will be performed by the *Receiver*, and this is the key of the Command Design Pattern.

Let's see an example. In our example we need to store some undo actions when the user is using the application, and when the user decides to perform the undo we can just use the invoker to run the commands. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) for our undo example:

[](http://www.dotnetlead.com/design-patterns/command/command2.PNG?attredirects=0)

* The *UndoCommand* class stores the location where the rectangle is supposed to move back.
* The *UndoPerformer* will move the rectangle back to its original position, taking the *UndoCommand* as the parameter.
* The *Invoker* stores the list of undo commands.

A Command Pattern says that "*encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command*".

It is also known as **Action or Transaction.**

#### Advantage of command pattern

* It separates the object that invokes the operation from the object that actually performs the operation.
* It makes easy to add new commands, because existing classes remain unchanged.

#### Usage of command pattern:

It is used:

* When you need parameterize objects according to an action perform.
* When you need to create and execute requests at different times.
* When you need to support rollback, logging or transaction functionality.

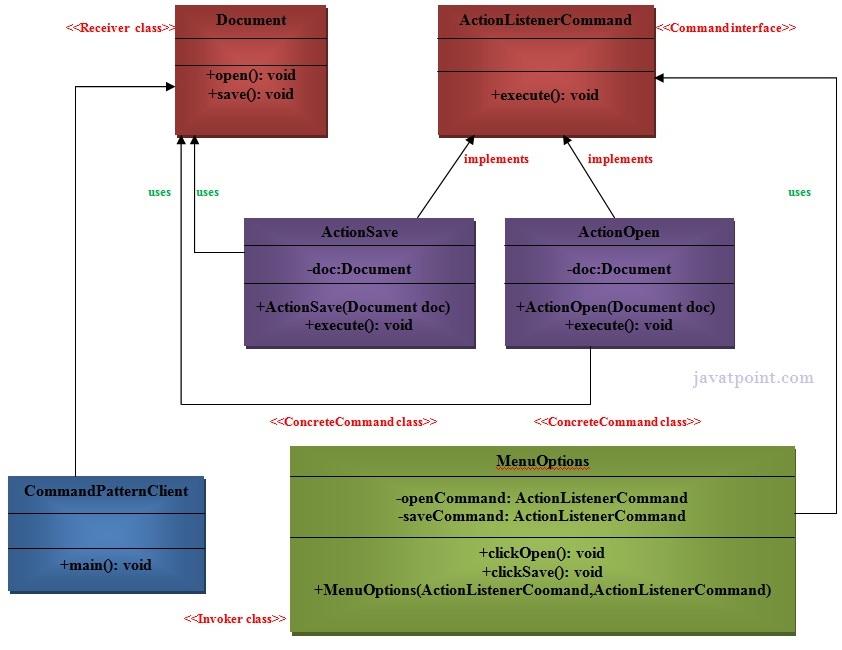
### Example of command pattern

Let's understand the example of adapter design pattern by the above UML diagram.

#### UML for command pattern:

##### These are the following participants of the Command Design pattern:

* **Command** This is an interface for executing an operation.
* **ConcreteCommand** This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
* **Client** This class creates the ConcreteCommand class and associates it with the receiver.
* **Invoker** This class asks the command to carry out the request.
* **Receiver** This class knows to perform the operation.



#### Implementation of above UML:

#### Step 1

Create a **ActionListernerCommand** interface that will act as a Command.

1. public interface ActionListenerCommand {
2. public void execute();
3. }

#### Step 2

Create a **Document** class that will act as a Receiver.

File: Document.java

1. public class Document {
2. public void open(){
3. System.out.println("Document Opened");
4. }
5. public void save(){
6. System.out.println("Document Saved");
7. }
8. }

#### Step 3

Create a **ActionOpen** class that will act as an ConcreteCommand.

File: ActionOpen.java

1. public class ActionOpen implements ActionListenerCommand{
2. private Document doc;
3. public ActionOpen(Document doc) {
4. this.doc = doc;
5. }
6. @Override
7. public void execute() {
8. doc.open();
9. }
10. }

#### Step 4

Create a **ActionSave** class that will act as an ConcreteCommand.

File: AdapterPatternDemo.java

1. public class ActionSave implements ActionListenerCommand{
2. private Document doc;
3. public ActionSave(Document doc) {
4. this.doc = doc;
5. }
6. @Override
7. public void execute() {
8. doc.save();
9. }
10. }

#### Step 5

Create a **MenuOptions** class that will act as an Invoker.

File: ActionSave.java

1. public class ActionSave implements ActionListenerCommand{
2. private Document doc;
3. public ActionSave(Document doc) {
4. this.doc = doc;
5. }
6. @Override
7. public void execute() {
8. doc.save();
9. }
10. }

#### Step 6

Create a **CommanPatternClient** class that will act as a Client.

File: AdapterPatternDemo.java

1. public class CommandPatternClient {
2. public static void main(String[] args) {
3. Document doc = new Document();
5. ActionListenerCommand clickOpen = new ActionOpen(doc);
6. ActionListenerCommand clickSave = new ActionSave(doc);
8. MenuOptions menu = new MenuOptions(clickOpen, clickSave);
10. menu.clickOpen();
11. menu.clickSave();
12. }
13. }

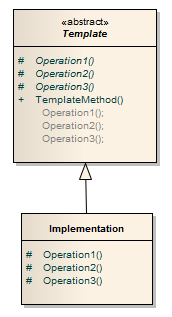
#### Output

1. Document Opened
2. Document Saved

**Template method:**

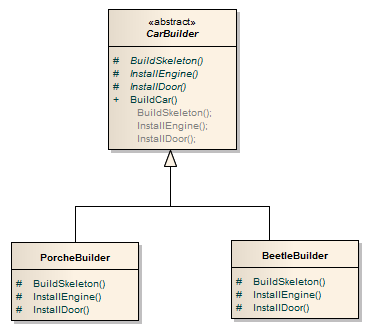
|  |  |
| --- | --- |
| The **Template Method Design Pattern** allows you to declare a general logic at the parent class so that all the child classes can use the general logic. To give an example, the general way of building a car is to build the skeleton first, followed by the installation of the engine, and finally the installation of the doors. This general sequence of building the car is the template method, since you will always need to use this logic no matter what types of cars you are building. |  |

The key to the template method pattern is that we put the general logic at the abstract parent class, and let the child class define the specifics. Let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the template method pattern, then we will look at an example to see how it works. The [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Template Method pattern is shown below:

[](http://www.dotnetlead.com/design-patterns/template-method/template1.png?attredirects=0)

* The *Template* class is the abstract parent class that contains the general logic.
  + It has the *Operation1, Operation2,* and *Operation3* methods, which are method stubs for the child classes to implement.
  + It has the *TemplateMethod* method, which contains the general logic. In the diagram it shows that the general logic calls *Operation1,* followed by *Operation2*, and finally *Operation3*.
* The *Implementation* class is the child class of the *Template* class. You will have multiple *Implementation* classes where each has its own specific implementation.
  + It has the *Operation1, Operation2,* and *Operation3* methods that contains the actual implementation of the methods.

Let's see an example. In the example we will build 2 types of cars. One is a Porsche, the other is a VW Beetle. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) for the example:

[](http://www.dotnetlead.com/design-patterns/template-method/template2.png?attredirects=0)

* The *CarBuilder* class is the parent abstract class that contains the general logic.
  + It has the *BuildCar* method, this is the template method that contains the general logic, which specifies that it must build the skeleton of the car first, followed by the installation of the engine, and finally the installation of the doors.
  + It has the *BuildSkeleton, InstallEngine,* and *InstallDoor* method stubs for the child classes to implement.
* The *PorcheBuilder* and the *BeetleBuilder* classes are the child classes of the *CarBuilder.* Each of the child classes will specify how to build their own skeleton, the engine, and the doors.
  + It has the *BuildSkeleton, InstallEngine,* and *InstallDoor* methods that contains the actual implementation of the methods.

With the template method pattern in place, the client code (calling code) can just be:

   CarBuilder c = new BeetleBuilder();

   c.BuildCar();

And it will build a Beetle with the specifications of a Beetle.

In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behavior pattern category.

## Implementation

We are going to create a *Game* abstract class defining operations with a template method set to be final so that it cannot be overridden. *Cricket* and *Football* are concrete classes that extend *Game* and override its methods.

*TemplatePatternDemo*, our demo class, will use *Game* to demonstrate use of template pattern.



## Step 1

Create an abstract class with a template method being final.

*Game.java*

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

//template method

public final void play(){

//initialize the game

initialize();

//start game

startPlay();

//end game

endPlay();

}

}

## Step 2

Create concrete classes extending the above class.

*Cricket.java*

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

*Football.java*

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

## Step 3

Use the *Game*'s template method play() to demonstrate a defined way of playing game.

*TemplatePatternDemo.java*

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

System.out.println();

game = new Football();

game.play();

}

}

## Step 4

Verify the output.

Cricket Game Initialized! Start playing.

Cricket Game Started. Enjoy the game!

Cricket Game Finished!

Football Game Initialized! Start playing.

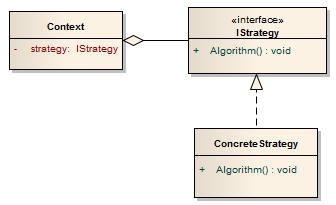
Football Game Started. Enjoy the game!

Football Game Finished!

**Strategy 🡪**

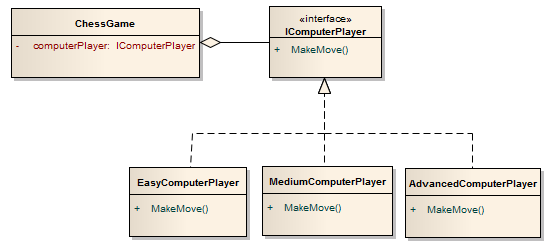
|  |  |
| --- | --- |
| The **Strategy Design Pattern** allows you to change the behavior of an application when given a context. The context is the outer shell the client code calls, and the behavior are defined by the strategy classes. The strategy pattern allows you to decouple the outer context from the internal behaviors. |  |

Let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the strategy pattern first, and then we will look at an example to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the strategy pattern:

[](http://www.dotnetlead.com/design-patterns/strategy/strategy1.png?attredirects=0)

* The *IStrategy* interface defines the behaviors supported.
  + It has the *Algorithm* method that specifies the behavior.
* The *ConcreteStrategy* class are classes that implements the behaviors. You will have multiple *ConcreteStrategy* classes where each class implements a behavior.
  + It has the *Algorithm* method that implements the behavior.
* The *Context* class defines the situation to use the behaviors.
  + It has the *strategy* variable that points to the behavior of the Context.

Let’s see an example. We would like to create a chess game with different levels of difficulty. The computer will respond based on the level of difficulty chosen by the user. The [UML](http://www.dotnetlead.com/UML-Quick-Reference) for the chess game will be:

[](http://www.dotnetlead.com/design-patterns/strategy/strategy2.png?attredirects=0)

* The *IComputerPlayer* interface defines the behaviors supported.
  + It has the *MakeMove* method, which makes a move for the computer player.
* The *EasyComputerPlayer, MediumComputerPlayer, AdvancedComputerPlayer* are the different difficulty levels.
  + It has the *MakeMove* method that implements the move the computer makes.
* The *ChessGame* class defines the situation to use the computer players.
  + It has the *computerPlayer* variable that holds a *IComputerPlayer*, which can be any level of difficulties.

The benefit of the strategy pattern is it allows you to choose the behavior of the application at runtime. You only need to define the behaviors as strategy classes, and the client code can simply choose any of the classes to exhibit the behavior.

A comparison between the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the strategy pattern and the [state pattern](http://www.dotnetlead.com/design-patterns/state) shows striking similarities. Both pattern defines the behaviors as the concrete classes, and both pattern defines the context in which the behavior runs under. But there are clear differences. The strategy pattern lets the client code choose the behavior it needs, while the [state pattern](http://www.dotnetlead.com/design-patterns/state) uses the behavior to switch to other behaviors.

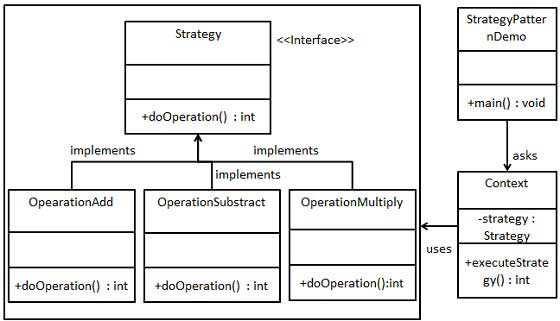
In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

## Implementation

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.

*StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



## Step 1

Create an interface.

*Strategy.java*

public interface Strategy {

public int doOperation(int num1, int num2);

}

## Step 2

Create concrete classes implementing the same interface.

*OperationAdd.java*

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

*OperationSubstract.java*

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

*OperationMultiply.java*

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

## Step 3

Create *Context* Class.

*Context.java*

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

## Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

*StrategyPatternDemo.java*

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

## Step 5

Verify the output.

10 + 5 = 15

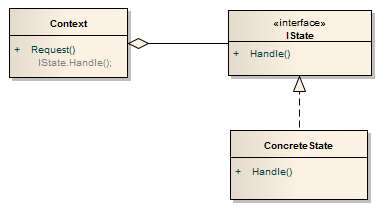
10 - 5 = 5

10 \* 5 = 50

**State design pattern 🡪**

|  |  |
| --- | --- |
| The **State Design Pattern** allows you to alter the behavior, or the state, of the object at runtime. Each behavior is represented by a state class, and the behavior can change from one state to another. |  |

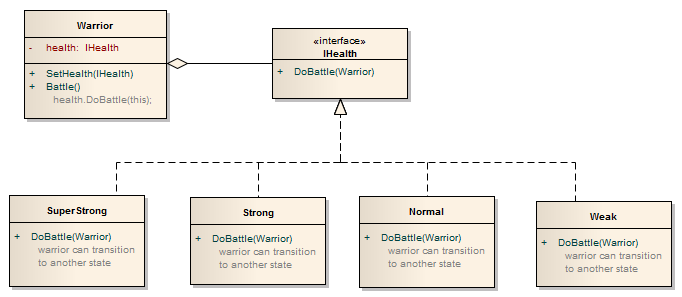
Let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the State Design Pattern first, then we will look at an example to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the state pattern:

[](http://www.dotnetlead.com/design-patterns/state/state1.png?attredirects=0)

* The *IState* interface defines the behaviors that the states can handle.
  + It has the *Handle* method that specifies the behavior of the state.
* The *ConcreteState* class are the classes that represents different behaviors. You will have multiple *ConcreteState* classes where each class will have its own implementation of behavior.
  + It has the *Handle* method that implements the state behavior, and it is within this method that you can change from one state to another.
* The *Context* class is the environment in which the states operates in.
  + It has the *Request* method that calls the *IState*'s methods.

Let’s see an example. Assuming you are writing a game where the player is a warrior that goes into different battles. Based on the outcome of each battle the warrior can become strong, super strong, normal, or weak. These will be the different states that the warrior can be in. Also the warrior is more likely to transition to certain states based on the state he was in before he goes into the battle. If the warrior is in the strong state before he goes into the battle, he is more likely to become super strong at the end of the battle than if he started out weak.

Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of our example:

[](http://www.dotnetlead.com/design-patterns/state/state2.png?attredirects=0)

* The *IHealth* interface defines methods for the states.
  + It has the *DoBattle* method when it's in a certain state of health.
* The *SupreStrong, Strong, Normal, and Weak* classes are all the health states, or behavior, that the warrior can be in.
  + It has the *DoBattle* method implementation, and it is here where the warrior can move from one health state to another.
* The *Warrior* class is the player, or the environment the health states operates in. The warrior can go into battle with a certain health state.
  + It has the *Battle* method, where it calls the *DoBattle* method of its *health* variable.
  + It has the *SetHealth* method that can set the health of the warrior.

One can also take the state pattern another step further by creating the *IWarrior* interface and decouple the relationship between the *Warrior* class and the *IHealth* interface. This way you can have different types of warrior where each warrior type will transition to different health differently. But for the purpose of demonstrating the state pattern only we will not do this in the implementation code below.

**A comparison between the** [**UML**](http://www.dotnetlead.com/UML-Quick-Reference) **of the state pattern and the** [**strategy pattern**](http://www.dotnetlead.com/design-patterns/strategy) **shows striking similarities. Both pattern defines the behaviors as the concrete classes, and both pattern defines the context in which the behavior runs under. But there are clear differences. The state pattern uses the behavior to switch to other behaviors, while the** [**strategy pattern**](http://www.dotnetlead.com/design-patterns/strategy) **lets the client code choose the behavior it needs.**

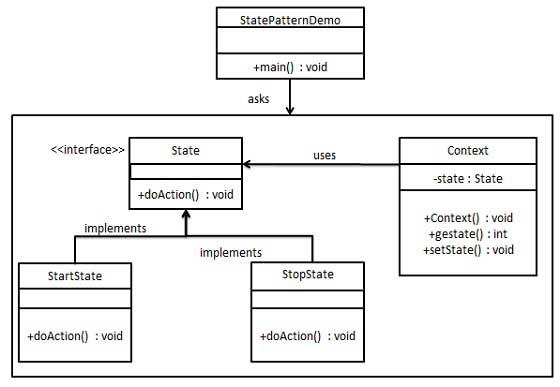
In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.

In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

## Implementation

We are going to create a *State* interface defining an action and concrete state classes implementing the *State* interface. *Context* is a class which carries a State.

*StatePatternDemo*, our demo class, will use *Context* and state objects to demonstrate change in Context behavior based on type of state it is in.



## Step 1

Create an interface.

*State.java*

public interface State {

public void doAction(Context context);

}

## Step 2

Create concrete classes implementing the same interface.

*StartState.java*

public class StartState implements State {

public void doAction(Context context) {

System.out.println("Player is in start state");

context.setState(this);

}

public String toString(){

return "Start State";

}

}

*StopState.java*

public class StopState implements State {

public void doAction(Context context) {

System.out.println("Player is in stop state");

context.setState(this);

}

public String toString(){

return "Stop State";

}

}

## Step 3

Create *Context* Class.

*Context.java*

public class Context {

private State state;

public Context(){

state = null;

}

public void setState(State state){

this.state = state;

}

public State getState(){

return state;

}

}

## Step 4

Use the *Context* to see change in behaviour when *State* changes.

*StatePatternDemo.java*

public class StatePatternDemo {

public static void main(String[] args) {

Context context = new Context();

StartState startState = new StartState();

startState.doAction(context);

System.out.println(context.getState().toString());

StopState stopState = new StopState();

stopState.doAction(context);

System.out.println(context.getState().toString());

}

}

## Step 5

Verify the output.

Player is in start state

Start State

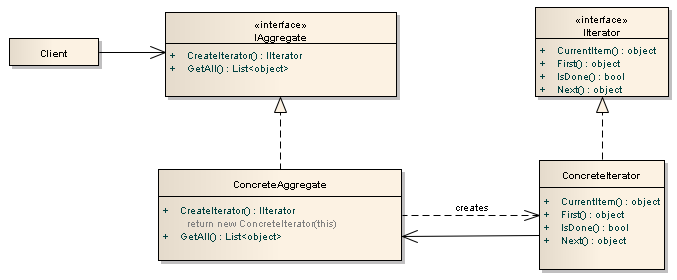
Player is in stop state

Stop State

**Iterator design pattern🡪**

|  |  |
| --- | --- |
| The **Iterator Design Pattern** allows you abstract out the details of traversing collections. For example, you may different types of collections in your applications, such as an array, a linked list, or a generic dictionary. For whichever the types of collections you have, you will need to traverse, or iterate through the items in the collections.  The actual implementation on how to traverse different types of collections will be different, yet the client code(calling code) should not be concerned about the details of the implementations. The iterator pattern helps you to hide such details and provide a generic interface for the client to traverse different types of collections. |  |

Let's look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the iterator pattern first, then we will look at some code to see the benefit that it brings. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Iterator Design Pattern:

[](http://www.dotnetlead.com/design-patterns/iterator/iterator.PNG?attredirects=0)

* The *IIterator* interface defines all the methods needed to traverse the collection.
* The *ConcreteIterator* class implements the *IIterator* interface and has the actual implementations on how to traverse the collection.
* The *IAggregate* interface defines the methods for the client. The methods that it defines allows the client code not to be bothered with the details on how the collection is traversed. It has the *GetAll* method that the client can call.
* The *ConcreteAggregate* class implements the *IAggregate* interface and is the class that creates the *ConcreteIterator*.

With the iterator pattern in place, the client code(calling code) can just be:

IAggregate a = new ConcreteAggregate();

List list = a.GetAll();  //gets the entire collection

Notice that the client code accesses only the *IAggregate* interface and does not need to know how the collection is traversed.

According to GoF, Iterator Pattern is used **"to access the elements of an aggregate object sequentially without exposing its underlying implementation".**

The Iterator pattern is also known as **Cursor.**

In collection framework, we are now using Iterator that is preferred over Enumeration.

#### java.util.Iterator interface uses Iterator Design Pattern.

#### Advantage of Iterator Pattern

* It supports variations in the traversal of a collection.
* It simplifies the interface to the collection.

#### Usage of Iterator Pattern:

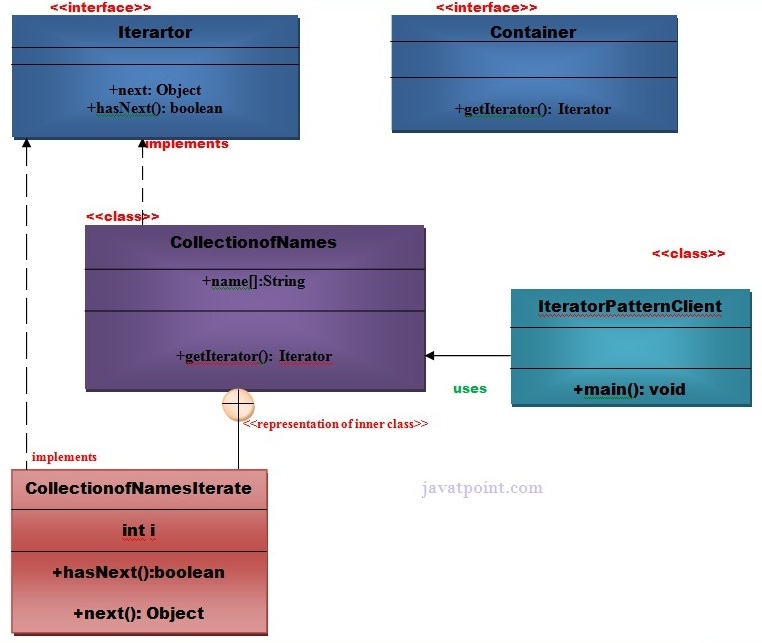
It is used:

* When you want to access a collection of objects without exposing its internal representation.
* When there are multiple traversals of objects need to be supported in the collection.

### Example of Iterator Pattern

Let's understand the example of iterator pattern pattern by the above UML diagram.

#### UML for Iterator Pattern:



#### Implementation of above UML

#### Step 1

Create a **Iterartor** interface.

1. public interface Iterator {
2. public boolean hasNext();
3. public Object next();
4. }

#### Step 2

Create a **Container** interface.

1. public interface Container {
2. public Iterator getIterator();
3. }// End of the Iterator interface.

#### Step 3

Create a **CollectionofNames** class that will implement **Container** interface.

File: CollectionofNames.java

1. public class CollectionofNames implements Container {
2. public String name[]={"Ashwani Rajput", "Soono Jaiswal","Rishi Kumar","Rahul Mehta","Hemant Mishra"};
4. @Override
5. public Iterator getIterator() {
6. return new CollectionofNamesIterate() ;
7. }
8. private class CollectionofNamesIterate implements Iterator{
9. int i;
10. @Override
11. public boolean hasNext() {
12. if (i<name.length){
13. return true;
14. }
15. return false;
16. }
17. @Override
18. public Object next() {
19. if(this.hasNext()){
20. return name[i++];
21. }
22. return null;
23. }
24. }
25. }
26. }

#### Step 4

Create a **IteratorPatternDemo** class.

File: IteratorPatternDemo.java

1. public class IteratorPatternDemo {
2. public static void main(String[] args) {
3. CollectionofNames cmpnyRepository = new CollectionofNames();
5. for(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){
6. String name = (String)iter.next();
7. System.out.println("Name : " + name);
8. }
9. }
10. }

[download this example](https://www.javatpoint.com/designpattern/designpatternexample/iteratorpattern.zip)

#### Output

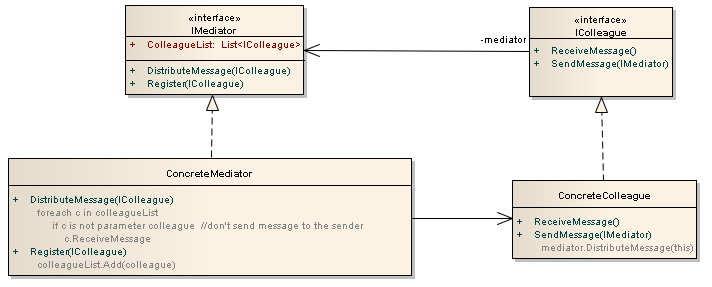
1. Name : Ashwani Rajput
2. Name : Soono Jaiswal
3. Name : Rishi Kumar
4. Name : Rahul Mehta
5. Name : Hemant Mishra

**Mediator design pattern 🡪**

|  |  |
| --- | --- |
| The **Mediator Design Pattern** allows you to decouple the direct communication between objects by introducing a middle object, the mediator, that facilitates the communication between the objects. Imagine you have a system where numerous objects communicate with each other by holding the reference to other objects. As the number of object grows and the references to other objects increases the system becomes hard to maintain. The mediator pattern is designed to solve this problem. |  |

The mediator is the communication center for the objects. When an object needs to communicate to another object, it does not call the other object directly. Instead, it calls the mediator object whose main duty is to route the messages to the destination object. It allows the developers not having to manage the links between the objects.

Let's look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the mediator pattern first, then we will look at some code to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Mediator Design Pattern:

[](http://www.dotnetlead.com/design-patterns/mediator/mediator.PNG?attredirects=0)

The left side is the mediator, the object that distributes the messages. The right side are the participants. The official [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the mediator pattern calls the participants as Colleagues, it's just a different terminology.

* The *IMediator* interface defines the properties and the methods that all mediators must support:
  + *ColleagueList* -- This is the list of the registered participants
  + *DistributeMessage(IColleague)* -- Sends the messages from the sender to all the participants
  + *Register(IColleague)* -- Register the participant to receive the message from the mediator
* The *ConcreteMediator* is the mediator class:
  + *DistributeMessage(IColleague)* -- Sends the message to the participants. It does not send the message back to the sender.
  + *Register(IColleague)* -- Register the participant to receive the message from the mediator
* The *IColleague* interface defines the methods that all participants must support:
  + *SendMessage(IMediator)* -- Sends the message to the mediator
  + *ReceiveMessage()* -- Gets the message from the mediator
* The *ConcreteColleague* class are the participants:
  + *SendMessage(IMediator)* -- Sends the message to the mediator by passing itself to the mediator
  + *ReceiveMessage()* -- Gets the message from the mediator

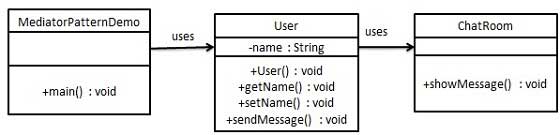
A comparison between the mediator pattern and the [observer pattern](http://www.dotnetlead.com/design-patterns/observer) shows some similarities and some clear differences. Both patterns facilitates the communication between objects, and both decouples the link between the sender and the receiver. The main difference is that in the mediator pattern there is the notion of the participants and they communicate with each other using the mediator as a central hub, whereas in the [observer pattern](http://www.dotnetlead.com/design-patterns/observer) there is a clear distinction between the sender and the receiver, and the receiver merely listens to the changes in the sender.

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

## Implementation

We are demonstrating mediator pattern by example of a chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes *ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages.

*MediatorPatternDemo*, our demo class, will use *User* objects to show communication between them.



## Step 1

Create mediator class.

*ChatRoom.java*

import java.util.Date;

public class ChatRoom {

public static void showMessage(User user, String message){

System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);

}

}

## Step 2

Create user class

*User.java*

public class User {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public User(String name){

this.name = name;

}

public void sendMessage(String message){

ChatRoom.showMessage(this,message);

}

}

## Step 3

Use the *User* object to show communications between them.

*MediatorPatternDemo.java*

public class MediatorPatternDemo {

public static void main(String[] args) {

User robert = new User("Robert");

User john = new User("John");

robert.sendMessage("Hi! John!");

john.sendMessage("Hello! Robert!");

}

}

## Step 4

Verify the output.

Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!

Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!

**Observer design pattern 🡪**

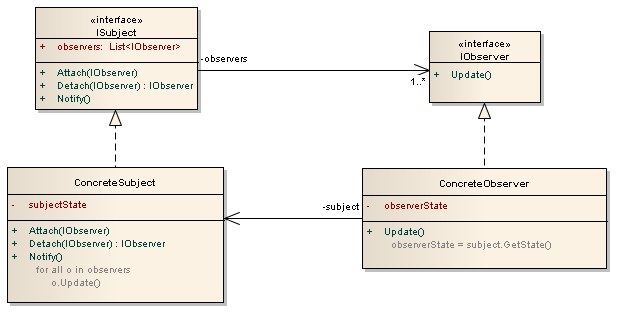
|  |  |  |
| --- | --- | --- |
|  | The **Observer Design Pattern** allows you to have a publisher-subscriber framework where a change to a publisher will notify all of its subscribers automatically.  The subscribers are registered to the publisher so that when a change occurs in the publisher all of the subscribers are notified. The publishers and the subscribers are decoupled through the use of interfaces so that the development of each can vary independently. |  |

There are 2 parts in the observer pattern:

1.       The first arethe ***subjects*. They are the publishers.** When a change occurs to a *subject* it should notify all of its subscribers.

2.       The second arethe ***observers*. They are the subscribers.**  They simply listen to the changes in the subjects.

The subjects are the publishers and the observers are the subscribers. It's just a different terminology. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Observer Design Pattern, the left part are the subjects, and the right part are the observers:

[](http://www.dotnetlead.com/design-patterns/observer/observer.PNG?attredirects=0)

* The *ISubject* is the interface that all publishers implement and has the following properties and methods:
  + *observers* -- List of observers that listen to the changes in the subject
  + *Attach(IObserver)* -- Adds an observer to listen to changes in the subject
  + *Detach(IObserver)* -- Remove an observer from listening changes in the subject
  + *Notify()* -- Send updates to all the observers that subscribed to it
* The *ConcreteSubject* is the publisher class and it implements the *ISubject* interface. Besides the implementation of the *ISubject* interface it also has the *subjectState* variable:
  + *subjectState* -- the variable that represents the state of the subject
* The *IObserver* is the interface that all subscribers implement and has the *Update* method:
  + *Update()* -- update the subscriber and is called by the subject (publisher)
* The *ConcreteObserver* is the subscriber class and it implements the *IObserver* interface. Below are its variables and methods:
  + *observerState* -- the variable that represents the state of the observer
  + *Update()* -- update the state of the observer. Notice that the method simply assigns the *observerState* variable from the subject's state. Therefore when a change to the subject's state occurs, the observer's state will become the same as the subject's state.

Notice in the *Update* method of the ConcreteObserver we assign the *observerState* variable as the subject's state and we only have one *observerState* variable. This means that the observer pattern is a one-to-many relationship, where one subject can have many observers listening to the subject's change but not vice versa.

A comparison between the [mediator pattern](http://www.dotnetlead.com/design-patterns/mediator) and the observer pattern shows some similarities and some clear differences. Both patterns facilitates the communication between objects, and both decouples the link between the sender and the receiver. The main difference is that in the [mediator pattern](http://www.dotnetlead.com/design-patterns/mediator) there is the notion of the participants and they communicate with each other using the mediator as a central hub, whereas in the observer pattern there is a clear distinction between the sender and the receiver, and the receiver merely listens to the changes in the sender.

While there are many different ways to implement the observer pattern, such as using delegates and events or the IObserver<T>, the concepts are all the same. That is, the observers are registered to listen to the changes in the subject and are notified when the subject changes. For the purpose of demonstrating the concept of the observer pattern we will not dig into the technicalities of the multiple ways to implement the pattern, but simply show how the observer pattern works by demonstrating the concept.

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

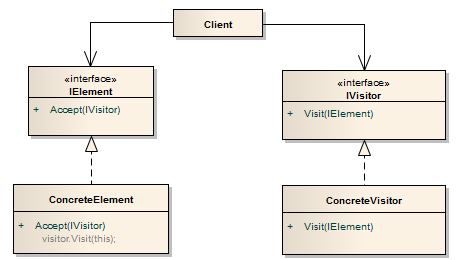
Octal String: 12

Binary String: 1010

**Visitor design pattern 🡪**

|  |  |
| --- | --- |
| The **Visitor Design Pattern** allows you to decouple the logics and the data structures and while applying the logics to the data structures. With this pattern you can build classes that focuses only on the data structures without knowing the logics that will be applied to the structure. At the same time, you can build classes that concentrate solely on the logics that will be applied to the structure without knowing what the structure looks like. The benefit is that the evolution of the logics and the structures can vary independently. |  |

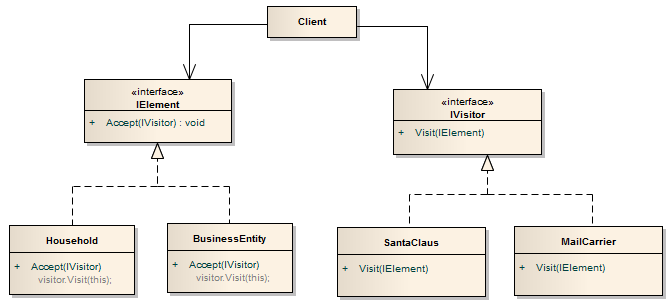
Let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the visitor pattern first, then we will look at an example to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Visitor Design Pattern:

[](http://www.dotnetlead.com/design-patterns/visitor/visitor1.png?attredirects=0)

* The left hand side are the classes for the structures
  + The *IElement* interface defines the elements in the structure.
    - It has the *Accept* method that takes in an *IVisitor*.
  + The *ConcreteElement* class implements the *IElement* interface. You will have multiple types of *ConcreteElement* classes where each represents different structure.
    - It has the *Accept* method that takes in an *IVisitor* that will apply the logic to the element.
* The right hand side are the classes for the logics
  + The *IVisitor* interface defines the logics supported.
    - It has the *Visit* method that takes in an *IElement*.
  + The *ConcreteVisitor* class implements the *IVisitor* interface. You will have multiple types of *ConcreteVisitor* classes where each represents different logic.
    - It has the *Visit* method that takes in an *IElement* that will apply the logic to the element.

Let's see an example. We will define 2 types of visitors, *SantaClaus* and *MailCarrier*. They both contain logics that will be applied to the structure.

We will define 2 types of structures, *Household* and *BusinessEntity*. They are the data structures where the visitors can apply their logics to. Both the *Household* and the *BusinessEntity* can be visited by *SantaClaus* or the *MailCarrier*.   
  
The [UML](http://www.dotnetlead.com/UML-Quick-Reference) of our example is shown below:

[](http://www.dotnetlead.com/design-patterns/visitor/visitor2.png?attredirects=0)

* The left hand side are the classes for the structures
  + The *IElement* interface defines the elements in the structure.
    - It has the *Accept* method that takes in an *IVisitor*.
  + The *HouseHold* and the *BusinessEntity* class implements the *IElement* interface.
    - They have the *Accept* method that takes in an *IVisitor* that will apply the logic to the element.
* The right hand side are the classes for the logics
  + The *IVisitor* interface defines the logics supported.
    - It has the *Visit* method that takes in an *IElement*.
  + The *SantaClaus* and the *MailCarrier* class implements the *IVisitor* interface.
    - They have the *Visit* method that takes in an *IElement* that will apply the logic to the element.

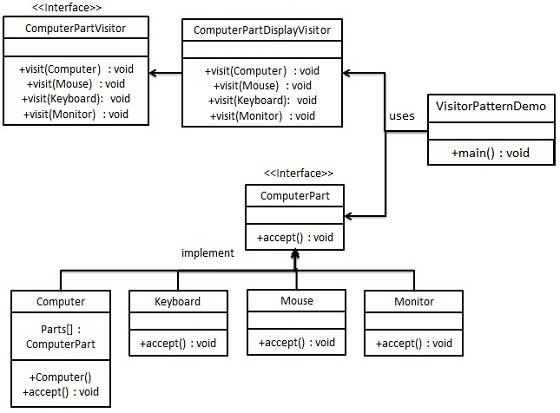
You can extend the visitor pattern by applying the [Composite Pattern](http://www.dotnetlead.com/design-patterns/composite) to the structure part of the visitor pattern, essentially forming a tree using the *IElement* as the interface for the tree structure where all the visitors can apply their logic to the tree structure. But for the purpose of demonstrating the visitor pattern only we will not make this extension in our implementation code.

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.

## 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.

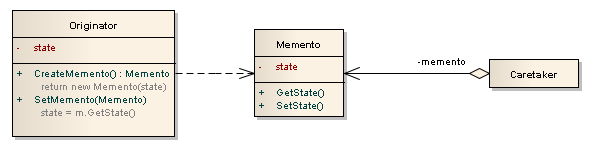
Displaying Monitor.

Displaying Computer.

**Memento design pattern 🡪**

|  |  |  |
| --- | --- | --- |
|  | The **Memento Design Pattern** allows you to save historical states of an object and restore the object back from the historical states. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later. An example are the checkpoints saved in a video game where the user is allowed to go back to the stages that they have already conquered. Another example are the undo operations in a word processing application. |  |

Let’s look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the memento pattern first, then we will look at some code to see how it works. Below is the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Memento Design Pattern:

[](http://www.dotnetlead.com/design-patterns/memento/memento.PNG?attredirects=0)

* The *Originator* class is the objects that will be saved and restored later:
  + The *state* variable contains information that represents the state of the *Originator* object. This is the variable that we save and restore.
  + The *CreateMemento* method is used to save the state of the *Originator.* It creates a *Memento* object by saving the *state* variable into the *Memento* object and return it. This is for recording the state of the *Originator*.
  + The *SetMemento* method restores the *Originator* by accepting a *Memento* object, unpackage it, and sets its *state* variable using the *state* variable from the *Memento*. This is for restoring the state of the *Originator* using the information that was previously saved in the *Memento*.
* The *Memento* class stores the historical information of the *Originator*. The information is stored in its *state* variable.
* The *Caretaker* class manages the list of *Memento*. This is the class for the client code to access.

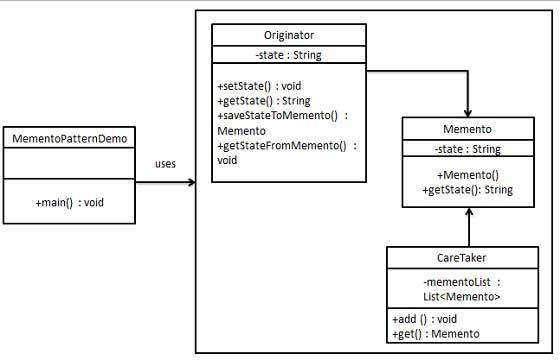
The key to the Memento Design Pattern is that the client code will never access the *Memento* object, all of the interactions are done through the *Caretaker* class. The client code does not need to be concerned about how the states are stored and retrieved.

Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.

## Implementation

Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. We have created classes *Memento*, *Originator* and *CareTaker*.

*MementoPatternDemo*, our demo class, will use *CareTaker* and *Originator* objects to show restoration of object states.



## Step 1

Create Memento class.

*Memento.java*

public class Memento {

private String state;

public Memento(String state){

this.state = state;

}

public String getState(){

return state;

}

}

## Step 2

Create Originator class

*Originator.java*

public class Originator {

private String state;

public void setState(String state){

this.state = state;

}

public String getState(){

return state;

}

public Memento saveStateToMemento(){

return new Memento(state);

}

public void getStateFromMemento(Memento memento){

state = memento.getState();

}

}

## Step 3

Create CareTaker class

*CareTaker.java*

import java.util.ArrayList;

import java.util.List;

public class CareTaker {

private List<Memento> mementoList = new ArrayList<Memento>();

public void add(Memento state){

mementoList.add(state);

}

public Memento get(int index){

return mementoList.get(index);

}

}

## Step 4

Use *CareTaker* and *Originator* objects.

*MementoPatternDemo.java*

public class MementoPatternDemo {

public static void main(String[] args) {

Originator originator = new Originator();

CareTaker careTaker = new CareTaker();

originator.setState("State #1");

originator.setState("State #2");

careTaker.add(originator.saveStateToMemento());

originator.setState("State #3");

careTaker.add(originator.saveStateToMemento());

originator.setState("State #4");

System.out.println("Current State: " + originator.getState());

originator.getStateFromMemento(careTaker.get(0));

System.out.println("First saved State: " + originator.getState());

originator.getStateFromMemento(careTaker.get(1));

System.out.println("Second saved State: " + originator.getState());

}

}

## Step 5

Verify the output.

Current State: State #4

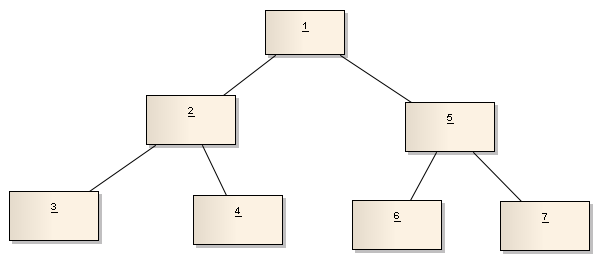
First saved State: State #2

Second saved State: State #3

**Interpreter design pattern 🡪**

|  |  |  |
| --- | --- | --- |
|  | The **Interpreter Design Pattern** allows you to build the rules as classes. This design pattern is very powerful in building the rules in a very logical manner. Most of the examples you see uses a programming language grammar as the rules because it's the easiest to demonstrate. However, once you realize the essence of the interpreter pattern you will see that you can apply the rules to any streams of inputs or objects.  The easiest way to understand how the interpreter pattern works is by looking at an example that will lead us to the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the interpreter pattern. |  |

There are 2 concepts to understand in the interpreter pattern.   
  
The first concept is the Depth First Search. In a depth first search you start off from the top of the tree, and you always go for the leftmost element as you go down the tree. You will gradually finish the left part of the tree first before you do the right part of the tree. Below is the diagram of the tree and the order to visit the nodes (from 1 to 7) using Depth First Search:

[](http://www.dotnetlead.com/design-patterns/interpreter/dfs.PNG?attredirects=0)

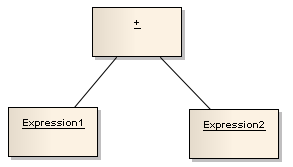
The second concept is the expression constructs of a language, they are simply the rules of the language. For example we can have the following rules in a programming language:

* AddExpression = Expression + Expression
* SubtractExpression = Expression - Expression
* Expression = NumberExpression | AddExpression | SubtractExpression

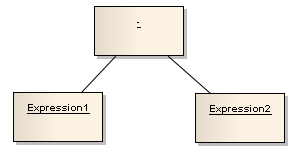
We can use these rules to give us a variety of ways to evaluate a statement:

* Example 1:
  + Expression = (10 - 2) + 3 = (SubtractExpression) + NumberExpression = Expression + Expression = AddExpression = 11
* Example 2:
  + Expression = (10 + 5) - (8 - 2) = (AddExpression) - (SubtractExpression) = Expression - Expression = SubtractExpression = 9

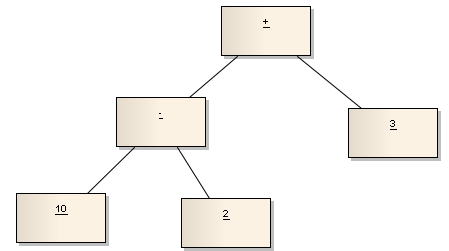
With the understanding of the Depth First Search and the rules of a language, we can build trees that represent the rules. The AddExpression tree is shown below:

[](http://www.dotnetlead.com/design-patterns/interpreter/addExp.PNG?attredirects=0)

Similarly, the SubtractExpression tree will be:

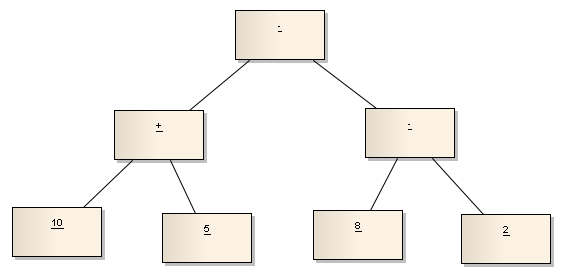
[](http://www.dotnetlead.com/design-patterns/interpreter/subExp.PNG?attredirects=0)

Taking Example1 as an example, the tree will be:

[](http://www.dotnetlead.com/design-patterns/interpreter/ex1.PNG?attredirects=0)

and if we traverse the tree above using Depth First Search, it will be: **+  -  10  2  3**

The tree for Example2 will be:

[](http://www.dotnetlead.com/design-patterns/interpreter/ex2.PNG?attredirects=0)

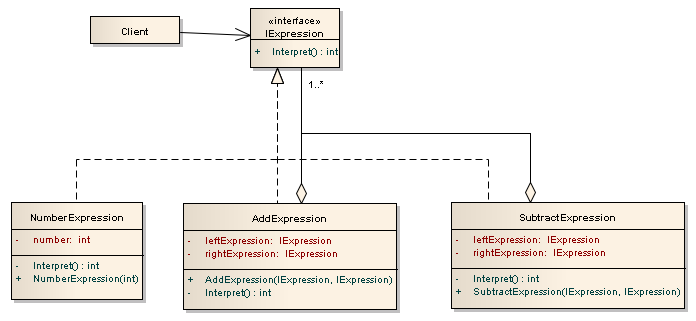
and if we traverse the tree above using Depth First Search, it will be:  **-  +  10  5  -  8  2**

The depth first search strings are the Reverse Polish Notations, and are commonly used in devices such as the financial calculators. The strings are the user inputs that will be evaluated using the rules of the language to calculate the results.

The interpreter design pattern allow us to take the rules of a language and build them as classes. In the example we have the following classes:

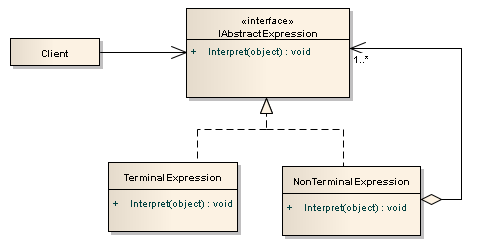
* *AddExpression*
* *SubtractExpression*
* *NumberExpression*

We can then build the [UML](http://www.dotnetlead.com/UML-Quick-Reference) for the example as shown below:

[](http://www.dotnetlead.com/design-patterns/interpreter/interpreter1.PNG?attredirects=0)

* The *IExpression* interface requires all expression classes to have the *Interpret* method, which means that all expressions can be interpreted.
* The *NumberExpression* class implements the *IExpression* interface, and since it's just a representation of a number it does not contain other expressions. Expressions that does not contain other expressions are called the TerminalExpressions.
* The *AddExpression* and *SubtractExpression* class implements the *IExpression* interface and contains other expressions. Expressions that contains other expressions are called the NonTerminalExpressions.

Now we can look at the [UML](http://www.dotnetlead.com/UML-Quick-Reference) of the Interpreter Design Pattern, which is the same as what we have in our example:

[](http://www.dotnetlead.com/design-patterns/interpreter/interpreter2.PNG?attredirects=0)

An Interpreter Pattern says that **"to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".**

Basically the Interpreter pattern has limited area where it can be applied. We can discuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions that is why it is not frequently used.

This pattern can applied for parsing the expressions defined in simple grammars and sometimes in simple rule engines.

#### SQL Parsing uses interpreter design pattern.

#### Advantage of Interpreter Pattern

* It is easier to change and extend the grammar.
* Implementing the grammar is straightforward.

#### Usage of Interpreter pattern:

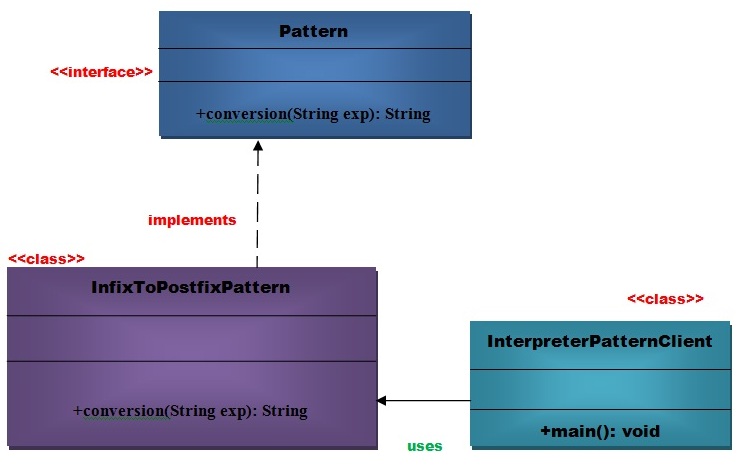
It is used:

* When the grammar of the language is not complicated.
* When the efficiency is not a priority.

### Example of Interpreter Pattern

Let's understand the example of Interpreter Pattern by the above UML diagram.

#### UML for Interpreter Pattern:



#### Implementation of above UML

#### Step 1

Create a **Pattern** interface.

1. public interface Pattern {
2. public String conversion(String exp);
3. }

#### Step 2

Create a **InfixToPostfixPattern** class that will allow what kind of pattern you want to convert.

File: InfixToPostfixPattern.java

1. import java.util.Stack;
2. public class InfixToPostfixPattern implements Pattern{
3. @Override
4. public String conversion(String exp) {
5. int priority = 0;// for the priority of operators.
6. String postfix = "";
7. Stack<Character> s1 = new Stack<Character>();
8. for (int i = 0; i < exp.length(); i++)
9. {
10. char ch = exp.charAt(i);
11. if (ch == '+' || ch == '-' || ch == '\*' || ch == '/'||ch=='%')
12. {
13. // check the precedence
14. if (s1.size() <= 0)
15. s1.push(ch);
16. }
17. else
18. {
19. Character chTop = (Character) s1.peek();
20. if (chTop == '\*' || chTop == '/')
21. priority = 1;
22. else
23. priority = 0;
24. if (priority == 1)
25. {
26. if (ch == '\*' || ch == '/'||ch=='%')
27. {
28. postfix += s1.pop();
29. i--;
30. }
31. else
32. { // Same
33. postfix += s1.pop();
34. i--;
35. }
36. }
37. else
38. {
39. if (ch == '+' || ch == '-')
40. {
41. postfix += s1.pop();
42. s1.push(ch);
43. }
44. else
45. s1.push(ch);
46. }
47. }
48. }
49. else
50. {
51. postfix += ch;
52. }
53. }
54. int len = s1.size();
55. for (int j = 0; j < len; j++)
56. postfix += s1.pop();
57. return postfix;
59. }
60. }// End of the InfixToPostfixPattern class.

#### Step 3

Create a **InterpreterPatternClient** class that will use InfixToPostfix Conversion.

File: InterpreterPatternClient.java

1. public class InterpreterPatternClient {
2. public static void main(String[] args)
3. {
4. String infix = "a+b\*c";
6. InfixToPostfixPattern ip=new InfixToPostfixPattern();
8. String postfix = ip.conversion(infix);
9. System.out.println("Infix:   " + infix);
10. System.out.println("Postfix: " + postfix);
11. }
12. }

[download this example](https://www.javatpoint.com/designpattern/designpatternexample/interpreterpattern.zip)

#### Output

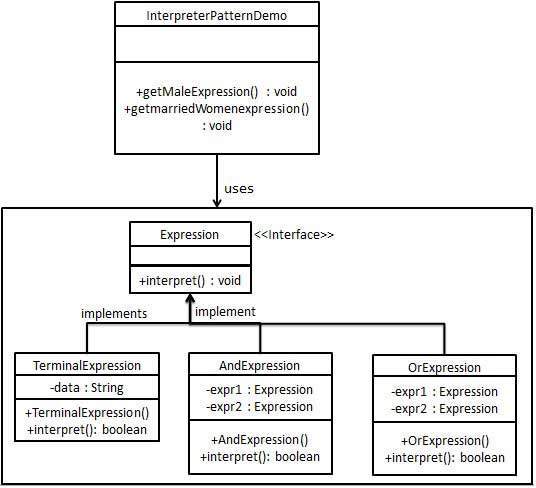
1. Infix:   a+b\*c
2. Postfix: abc\*+

Interpreter pattern provides a way to evaluate language grammar or expression. This type of pattern comes under behavioral pattern. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.

## Implementation

We are going to create an interface *Expression* and concrete classes implementing the *Expression* interface. A class *TerminalExpression* is defined which acts as a main interpreter of context in question. Other classes *OrExpression*, *AndExpression* are used to create combinational expressions.

*InterpreterPatternDemo*, our demo class, will use *Expression* class to create rules and demonstrate parsing of expressions.



## Step 1

Create an expression interface.

*Expression.java*

public interface Expression {

public boolean interpret(String context);

}

## Step 2

Create concrete classes implementing the above interface.

*TerminalExpression.java*

public class TerminalExpression implements Expression {

private String data;

public TerminalExpression(String data){

this.data = data;

}

@Override

public boolean interpret(String context) {

if(context.contains(data)){

return true;

}

return false;

}

}

*OrExpression.java*

public class OrExpression implements Expression {

private Expression expr1 = null;

private Expression expr2 = null;

public OrExpression(Expression expr1, Expression expr2) {

this.expr1 = expr1;

this.expr2 = expr2;

}

@Override

public boolean interpret(String context) {

return expr1.interpret(context) || expr2.interpret(context);

}

}

*AndExpression.java*

public class AndExpression implements Expression {

private Expression expr1 = null;

private Expression expr2 = null;

public AndExpression(Expression expr1, Expression expr2) {

this.expr1 = expr1;

this.expr2 = expr2;

}

@Override

public boolean interpret(String context) {

return expr1.interpret(context) && expr2.interpret(context);

}

}

## Step 3

*InterpreterPatternDemo* uses *Expression* class to create rules and then parse them.

*InterpreterPatternDemo.java*

public class InterpreterPatternDemo {

//Rule: Robert and John are male

public static Expression getMaleExpression(){

Expression robert = new TerminalExpression("Robert");

Expression john = new TerminalExpression("John");

return new OrExpression(robert, john);

}

//Rule: Julie is a married women

public static Expression getMarriedWomanExpression(){

Expression julie = new TerminalExpression("Julie");

Expression married = new TerminalExpression("Married");

return new AndExpression(julie, married);

}

public static void main(String[] args) {

Expression isMale = getMaleExpression();

Expression isMarriedWoman = getMarriedWomanExpression();

System.out.println("John is male? " + isMale.interpret("John"));

System.out.println("Julie is a married women? " + isMarriedWoman.interpret("Married Julie"));

}

}

## Step 4

Verify the output.

John is male? true

Julie is a married women? true

**MVC design pattern 🡪**

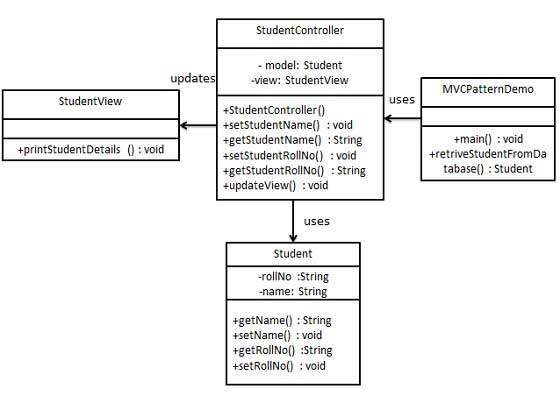
MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

* **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
* **View** - View represents the visualization of the data that model contains.
* **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

## Implementation

We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController* is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.

*MVCPatternDemo*, our demo class, will use *StudentController* to demonstrate use of MVC pattern.



## Step 1

Create Model.

*Student.java*

public class Student {

private String rollNo;

private String name;

public String getRollNo() {

return rollNo;

}

public void setRollNo(String rollNo) {

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

## Step 2

Create View.

*StudentView.java*

public class StudentView {

public void printStudentDetails(String studentName, String studentRollNo){

System.out.println("Student: ");

System.out.println("Name: " + studentName);

System.out.println("Roll No: " + studentRollNo);

}

}

## Step 3

Create Controller.

*StudentController.java*

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view){

this.model = model;

this.view = view;

}

public void setStudentName(String name){

model.setName(name);

}

public String getStudentName(){

return model.getName();

}

public void setStudentRollNo(String rollNo){

model.setRollNo(rollNo);

}

public String getStudentRollNo(){

return model.getRollNo();

}

public void updateView(){

view.printStudentDetails(model.getName(), model.getRollNo());

}

}

## Step 4

Use the *StudentController* methods to demonstrate MVC design pattern usage.

*MVCPatternDemo.java*

public class MVCPatternDemo {

public static void main(String[] args) {

//fetch student record based on his roll no from the database

Student model = retriveStudentFromDatabase();

//Create a view : to write student details on console

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

//update model data

controller.setStudentName("John");

controller.updateView();

}

private static Student retriveStudentFromDatabase(){

Student student = new Student();

student.setName("Robert");

student.setRollNo("10");

return student;

}

}

## Step 5

Verify the output.

Student:

Name: Robert

Roll No: 10

Student:

Name: John

Roll No: 10

**NULL object pattern🡪**

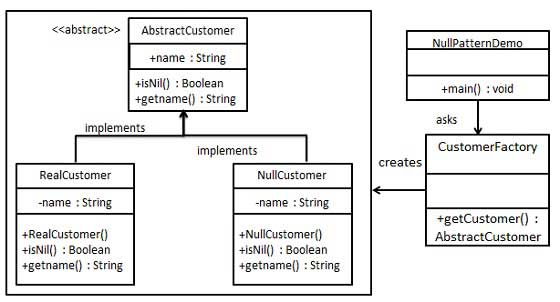
In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

In Null Object pattern, we create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implemention of this class and will be used seemlessly where we need to check null value.

## Implementation

We are going to create a *AbstractCustomer* abstract class defining opearations. Here the name of the customer and concrete classes extending the *AbstractCustomer* class. A factory class *CustomerFactory* is created to return either *RealCustomer* or *NullCustomer* objects based on the name of customer passed to it.

*NullPatternDemo*, our demo class, will use *CustomerFactory* to demonstrate the use of Null Object pattern.



## Step 1

Create an abstract class.

*AbstractCustomer.java*

public abstract class AbstractCustomer {

protected String name;

public abstract boolean isNil();

public abstract String getName();

}

## Step 2

Create concrete classes extending the above class.

*RealCustomer.java*

public class RealCustomer extends AbstractCustomer {

public RealCustomer(String name) {

this.name = name;

}

@Override

public String getName() {

return name;

}

@Override

public boolean isNil() {

return false;

}

}

*NullCustomer.java*

public class NullCustomer extends AbstractCustomer {

@Override

public String getName() {

return "Not Available in Customer Database";

}

@Override

public boolean isNil() {

return true;

}

}

## Step 3

Create *CustomerFactory* Class.

*CustomerFactory.java*

public class CustomerFactory {

public static final String[] names = {"Rob", "Joe", "Julie"};

public static AbstractCustomer getCustomer(String name){

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

if (names[i].equalsIgnoreCase(name)){

return new RealCustomer(name);

}

}

return new NullCustomer();

}

}

## Step 4

Use the *CustomerFactory* to get either *RealCustomer* or *NullCustomer* objects based on the name of customer passed to it.

*NullPatternDemo.java*

public class NullPatternDemo {

public static void main(String[] args) {

AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");

AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");

AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");

AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

System.out.println("Customers");

System.out.println(customer1.getName());

System.out.println(customer2.getName());

System.out.println(customer3.getName());

System.out.println(customer4.getName());

}

}

## Step 5

Verify the output.

Customers

Rob

Not Available in Customer Database

Julie

Not Available in Customer Database

**Business Delegate Pattern🡪**

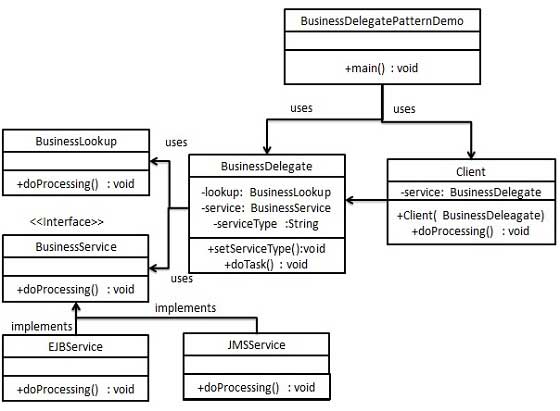
Business Delegate Pattern is used to decouple presentation tier and business tier. It is basically use to reduce communication or remote lookup functionality to business tier code in presentation tier code. In business tier we have following entities.

* **Client** - Presentation tier code may be JSP, servlet or UI java code.
* **Business Delegate** - A single entry point class for client entities to provide access to Business Service methods.
* **LookUp Service** - Lookup service object is responsible to get relative business implementation and provide business object access to business delegate object.
* **Business Service** - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

## Implementation

We are going to create a *Client*, *BusinessDelegate*, *BusinessService*, *LookUpService*, *JMSService* and *EJBService* representing various entities of Business Delegate patterns.

*BusinessDelegatePatternDemo*, our demo class, will use *BusinessDelegate* and *Client* to demonstrate use of Business Delegate pattern.



## Step 1

Create BusinessService Interface.

*BusinessService.java*

public interface BusinessService {

public void doProcessing();

}

## Step 2

Create concrete Service classes.

*EJBService.java*

public class EJBService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking EJB Service");

}

}

*JMSService.java*

public class JMSService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking JMS Service");

}

}

## Step 3

Create Business Lookup Service.

*BusinessLookUp.java*

public class BusinessLookUp {

public BusinessService getBusinessService(String serviceType){

if(serviceType.equalsIgnoreCase("EJB")){

return new EJBService();

}

else {

return new JMSService();

}

}

}

## Step 4

Create Business Delegate.

*BusinessDelegate.java*

public class BusinessDelegate {

private BusinessLookUp lookupService = new BusinessLookUp();

private BusinessService businessService;

private String serviceType;

public void setServiceType(String serviceType){

this.serviceType = serviceType;

}

public void doTask(){

businessService = lookupService.getBusinessService(serviceType);

businessService.doProcessing();

}

}

## Step 5

Create Client.

*Client.java*

public class Client {

BusinessDelegate businessService;

public Client(BusinessDelegate businessService){

this.businessService = businessService;

}

public void doTask(){

businessService.doTask();

}

}

## Step 6

Use BusinessDelegate and Client classes to demonstrate Business Delegate pattern.

*BusinessDelegatePatternDemo.java*

public class BusinessDelegatePatternDemo {

public static void main(String[] args) {

BusinessDelegate businessDelegate = new BusinessDelegate();

businessDelegate.setServiceType("EJB");

Client client = new Client(businessDelegate);

client.doTask();

businessDelegate.setServiceType("JMS");

client.doTask();

}

}

## Step 7

Verify the output.

Processing task by invoking EJB Service

Processing task by invoking JMS Service

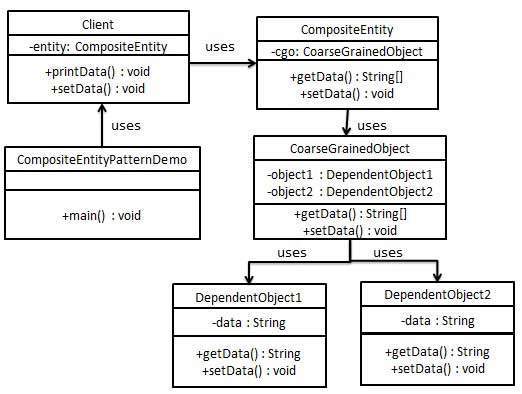
**Composite Entity pattern🡪**

Composite Entity pattern is used in EJB persistence mechanism. A Composite entity is an EJB entity bean which represents a graph of objects. When a composite entity is updated, internally dependent objects beans get updated automatically as being managed by EJB entity bean. Following are the participants in Composite Entity Bean.

* **Composite Entity** - It is primary entity bean. It can be coarse grained or can contain a coarse grained object to be used for persistence purpose.
* **Coarse-Grained Object** - This object contains dependent objects. It has its own life cycle and also manages life cycle of dependent objects.
* **Dependent Object** - Dependent object is an object which depends on coarse grained object for its persistence lifecycle.
* **Strategies** - Strategies represents how to implement a Composite Entity.

## Implementation

We are going to create *CompositeEntity* object acting as CompositeEntity. *CoarseGrainedObject* will be a class which contains dependent objects. *CompositeEntityPatternDemo*, our demo class will use *Client* class to demonstrate use of Composite Entity pattern.



## Step 1

Create Dependent Objects.

*DependentObject1.java*

public class DependentObject1 {

private String data;

public void setData(String data){

this.data = data;

}

public String getData(){

return data;

}

}

*DependentObject2.java*

public class DependentObject2 {

private String data;

public void setData(String data){

this.data = data;

}

public String getData(){

return data;

}

}

## Step 2

Create Coarse Grained Object.

*CoarseGrainedObject.java*

public class CoarseGrainedObject {

DependentObject1 do1 = new DependentObject1();

DependentObject2 do2 = new DependentObject2();

public void setData(String data1, String data2){

do1.setData(data1);

do2.setData(data2);

}

public String[] getData(){

return new String[] {do1.getData(),do2.getData()};

}

}

## Step 3

Create Composite Entity.

*CompositeEntity.java*

public class CompositeEntity {

private CoarseGrainedObject cgo = new CoarseGrainedObject();

public void setData(String data1, String data2){

cgo.setData(data1, data2);

}

public String[] getData(){

return cgo.getData();

}

}

## Step 4

Create Client class to use Composite Entity.

*Client.java*

public class Client {

private CompositeEntity compositeEntity = new CompositeEntity();

public void printData(){

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

System.out.println("Data: " + compositeEntity.getData()[i]);

}

}

public void setData(String data1, String data2){

compositeEntity.setData(data1, data2);

}

}

## Step 5

Use the *Client* to demonstrate Composite Entity design pattern usage.

*CompositeEntityPatternDemo.java*

public class CompositeEntityPatternDemo {

public static void main(String[] args) {

Client client = new Client();

client.setData("Test", "Data");

client.printData();

client.setData("Second Test", "Data1");

client.printData();

}

}

## Step 6

Verify the output.

Data: Test

Data: Data

Data: Second Test

Data: Data1

**Data access object pattern🡪**

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

* **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
* **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
* **Model Object or Value Object** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

## Implementation

We are going to create a *Student* object acting as a Model or Value Object.*StudentDao* is Data Access Object Interface.*StudentDaoImpl* is concrete class implementing Data Access Object Interface. *DaoPatternDemo*, our demo class, will use *StudentDao* to demonstrate the use of Data Access Object pattern.



## Step 1

Create Value Object.

*Student.java*

public class Student {

private String name;

private int rollNo;

Student(String name, int rollNo){

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

}

## Step 2

Create Data Access Object Interface.

*StudentDao.java*

import java.util.List;

public interface StudentDao {

public List<Student> getAllStudents();

public Student getStudent(int rollNo);

public void updateStudent(Student student);

public void deleteStudent(Student student);

}

## Step 3

Create concrete class implementing above interface.

*StudentDaoImpl.java*

import java.util.ArrayList;

import java.util.List;

public class StudentDaoImpl implements StudentDao {

//list is working as a database

List<Student> students;

public StudentDaoImpl(){

students = new ArrayList<Student>();

Student student1 = new Student("Robert",0);

Student student2 = new Student("John",1);

students.add(student1);

students.add(student2);

}

@Override

public void deleteStudent(Student student) {

students.remove(student.getRollNo());

System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");

}

//retrive list of students from the database

@Override

public List<Student> getAllStudents() {

return students;

}

@Override

public Student getStudent(int rollNo) {

return students.get(rollNo);

}

@Override

public void updateStudent(Student student) {

students.get(student.getRollNo()).setName(student.getName());

System.out.println("Student: Roll No " + student.getRollNo() + ", updated in the database");

}

}

## Step 4

Use the *StudentDao* to demonstrate Data Access Object pattern usage.

*DaoPatternDemo.java*

public class DaoPatternDemo {

public static void main(String[] args) {

StudentDao studentDao = new StudentDaoImpl();

//print all students

for (Student student : studentDao.getAllStudents()) {

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

//update student

Student student =studentDao.getAllStudents().get(0);

student.setName("Michael");

studentDao.updateStudent(student);

//get the student

studentDao.getStudent(0);

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

}

## Step 5

Verify the output.

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]

**Front controller pattern🡪**

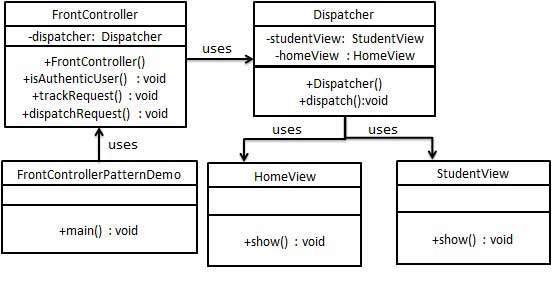
The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

* **Front Controller** - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
* **Dispatcher** - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.
* **View** - Views are the object for which the requests are made.

## Implementation

We are going to create a *FrontController* and *Dispatcher* to act as Front Controller and Dispatcher correspondingly. *HomeView* and *StudentView* represent various views for which requests can come to front controller.

*FrontControllerPatternDemo*, our demo class, will use *FrontController* to demonstrate Front Controller Design Pattern.



## Step 1

Create Views.

*HomeView.java*

public class HomeView {

public void show(){

System.out.println("Displaying Home Page");

}

}

*StudentView.java*

public class StudentView {

public void show(){

System.out.println("Displaying Student Page");

}

}

## Step 2

Create Dispatcher.

*Dispatcher.java*

public class Dispatcher {

private StudentView studentView;

private HomeView homeView;

public Dispatcher(){

studentView = new StudentView();

homeView = new HomeView();

}

public void dispatch(String request){

if(request.equalsIgnoreCase("STUDENT")){

studentView.show();

}

else{

homeView.show();

}

}

}

## Step 3

Create FrontController

*FrontController.java*

public class FrontController {

private Dispatcher dispatcher;

public FrontController(){

dispatcher = new Dispatcher();

}

private boolean isAuthenticUser(){

System.out.println("User is authenticated successfully.");

return true;

}

private void trackRequest(String request){

System.out.println("Page requested: " + request);

}

public void dispatchRequest(String request){

//log each request

trackRequest(request);

//authenticate the user

if(isAuthenticUser()){

dispatcher.dispatch(request);

}

}

}

## Step 4

Use the *FrontController* to demonstrate Front Controller Design Pattern.

*FrontControllerPatternDemo.java*

public class FrontControllerPatternDemo {

public static void main(String[] args) {

FrontController frontController = new FrontController();

frontController.dispatchRequest("HOME");

frontController.dispatchRequest("STUDENT");

}

}

## Step 5

Verify the output.

Page requested: HOME

User is authenticated successfully.

Displaying Home Page

Page requested: STUDENT

User is authenticated successfully.

Displaying Student Page

**Interpreter controller pattern:**

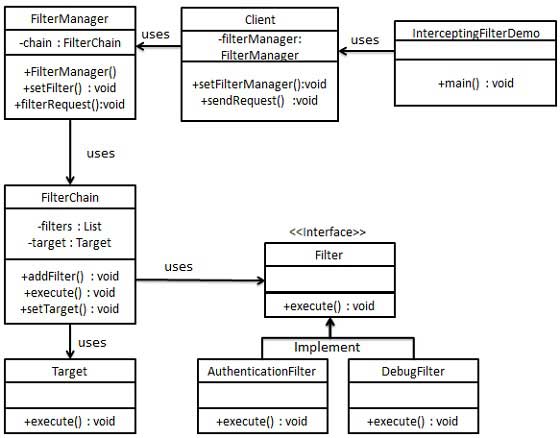
The intercepting filter design pattern is used when we want to do some pre-processing / post-processing with request or response of the application. Filters are defined and applied on the request before passing the request to actual target application. Filters can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

* **Filter** - Filter which will performs certain task prior or after execution of request by request handler.
* **Filter Chain** - Filter Chain carries multiple filters and help to execute them in defined order on target.
* **Target** - Target object is the request handler
* **Filter Manager** - Filter Manager manages the filters and Filter Chain.
* **Client** - Client is the object who sends request to the Target object.

## Implementation

We are going to create a *FilterChain*,*FilterManager*, *Target*, *Client* as various objects representing our entities.*AuthenticationFilter* and *DebugFilter* represent concrete filters.

*InterceptingFilterDemo*, our demo class, will use *Client* to demonstrate Intercepting Filter Design Pattern.



## Step 1

Create Filter interface.

*Filter.java*

public interface Filter {

public void execute(String request);

}

## Step 2

Create concrete filters.

*AuthenticationFilter.java*

public class AuthenticationFilter implements Filter {

public void execute(String request){

System.out.println("Authenticating request: " + request);

}

}

*DebugFilter.java*

public class DebugFilter implements Filter {

public void execute(String request){

System.out.println("request log: " + request);

}

}

## Step 3

Create Target

*Target.java*

public class Target {

public void execute(String request){

System.out.println("Executing request: " + request);

}

}

## Step 4

Create Filter Chain

*FilterChain.java*

import java.util.ArrayList;

import java.util.List;

public class FilterChain {

private List<Filter> filters = new ArrayList<Filter>();

private Target target;

public void addFilter(Filter filter){

filters.add(filter);

}

public void execute(String request){

for (Filter filter : filters) {

filter.execute(request);

}

target.execute(request);

}

public void setTarget(Target target){

this.target = target;

}

}

## Step 5

Create Filter Manager

*FilterManager.java*

public class FilterManager {

FilterChain filterChain;

public FilterManager(Target target){

filterChain = new FilterChain();

filterChain.setTarget(target);

}

public void setFilter(Filter filter){

filterChain.addFilter(filter);

}

public void filterRequest(String request){

filterChain.execute(request);

}

}

## Step 6

Create Client

*Client.java*

public class Client {

FilterManager filterManager;

public void setFilterManager(FilterManager filterManager){

this.filterManager = filterManager;

}

public void sendRequest(String request){

filterManager.filterRequest(request);

}

}

## Step 7

Use the *Client* to demonstrate Intercepting Filter Design Pattern.

*InterceptingFilterDemo.java*

public class InterceptingFilterDemo {

public static void main(String[] args) {

FilterManager filterManager = new FilterManager(new Target());

filterManager.setFilter(new AuthenticationFilter());

filterManager.setFilter(new DebugFilter());

Client client = new Client();

client.setFilterManager(filterManager);

client.sendRequest("HOME");

}

}

## Step 8

Verify the output.

Authenticating request: HOME

request log: HOME

Executing request: HOME

**Service locater pattern 🡪**

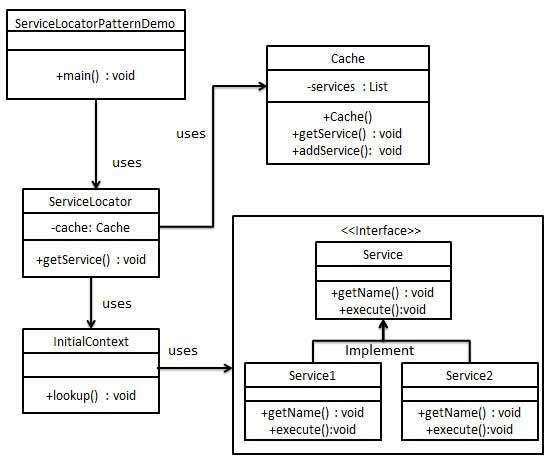
The service locator design pattern is used when we want to locate various services using JNDI lookup. Considering high cost of looking up JNDI for a service, Service Locator pattern makes use of caching technique. For the first time a service is required, Service Locator looks up in JNDI and caches the service object. Further lookup or same service via Service Locator is done in its cache which improves the performance of application to great extent. Following are the entities of this type of design pattern.

* **Service** - Actual Service which will process the request. Reference of such service is to be looked upon in JNDI server.
* **Context / Initial Context** - JNDI Context carries the reference to service used for lookup purpose.
* **Service Locator** - Service Locator is a single point of contact to get services by JNDI lookup caching the services.
* **Cache** - Cache to store references of services to reuse them
* **Client** - Client is the object that invokes the services via ServiceLocator.

## Implementation

We are going to create a *ServiceLocator*,*InitialContext*, *Cache*, *Service* as various objects representing our entities.*Service1* and *Service2* represent concrete services.

*ServiceLocatorPatternDemo*, our demo class, is acting as a client here and will use *ServiceLocator* to demonstrate Service Locator Design Pattern.



## Step 1

Create Service interface.

*Service.java*

public interface Service {

public String getName();

public void execute();

}

## Step 2

Create concrete services.

*Service1.java*

public class Service1 implements Service {

public void execute(){

System.out.println("Executing Service1");

}

@Override

public String getName() {

return "Service1";

}

}

*Service2.java*

public class Service2 implements Service {

public void execute(){

System.out.println("Executing Service2");

}

@Override

public String getName() {

return "Service2";

}

}

## Step 3

Create InitialContext for JNDI lookup

*InitialContext.java*

public class InitialContext {

public Object lookup(String jndiName){

if(jndiName.equalsIgnoreCase("SERVICE1")){

System.out.println("Looking up and creating a new Service1 object");

return new Service1();

}

else if (jndiName.equalsIgnoreCase("SERVICE2")){

System.out.println("Looking up and creating a new Service2 object");

return new Service2();

}

return null;

}

}

## Step 4

Create Cache

*Cache.java*

import java.util.ArrayList;

import java.util.List;

public class Cache {

private List<Service> services;

public Cache(){

services = new ArrayList<Service>();

}

public Service getService(String serviceName){

for (Service service : services) {

if(service.getName().equalsIgnoreCase(serviceName)){

System.out.println("Returning cached " + serviceName + " object");

return service;

}

}

return null;

}

public void addService(Service newService){

boolean exists = false;

for (Service service : services) {

if(service.getName().equalsIgnoreCase(newService.getName())){

exists = true;

}

}

if(!exists){

services.add(newService);

}

}

}

## Step 5

Create Service Locator

*ServiceLocator.java*

public class ServiceLocator {

private static Cache cache;

static {

cache = new Cache();

}

public static Service getService(String jndiName){

Service service = cache.getService(jndiName);

if(service != null){

return service;

}

InitialContext context = new InitialContext();

Service service1 = (Service)context.lookup(jndiName);

cache.addService(service1);

return service1;

}

}

## Step 6

Use the *ServiceLocator* to demonstrate Service Locator Design Pattern.

*ServiceLocatorPatternDemo.java*

public class ServiceLocatorPatternDemo {

public static void main(String[] args) {

Service service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

}

}

## Step 7

Verify the output.

Looking up and creating a new Service1 object

Executing Service1

Looking up and creating a new Service2 object

Executing Service2

Returning cached Service1 object

Executing Service1

Returning cached Service2 object

Executing Service2

**Transfer object pattern:**

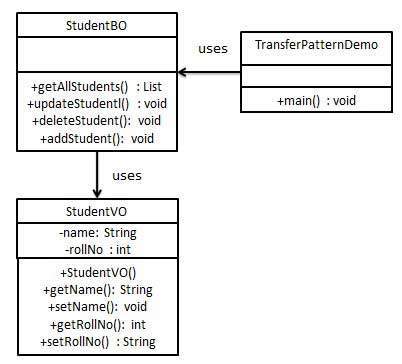
The Transfer Object pattern is used when we want to pass data with multiple attributes in one shot from client to server. Transfer object is also known as Value Object. Transfer Object is a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network. It does not have any behavior. Server Side business class normally fetches data from the database and fills the POJO and send it to the client or pass it by value. For client, transfer object is read-only. Client can create its own transfer object and pass it to server to update values in database in one shot. Following are the entities of this type of design pattern.

* **Business Object** - Business Service fills the Transfer Object with data.
* **Transfer Object** - Simple POJO having methods to set/get attributes only.
* **Client** - Client either requests or sends the Transfer Object to Business Object.

## Implementation

We are going to create a *StudentBO* as Business Object,*Student* as Transfer Object representing our entities.

*TransferObjectPatternDemo*, our demo class, is acting as a client here and will use *StudentBO* and *Student* to demonstrate Transfer Object Design Pattern.



## Step 1

Create Transfer Object.

*StudentVO.java*

public class StudentVO {

private String name;

private int rollNo;

StudentVO(String name, int rollNo){

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

}

## Step 2

Create Business Object.

*StudentBO.java*

import java.util.ArrayList;

import java.util.List;

public class StudentBO {

//list is working as a database

List<StudentVO> students;

public StudentBO(){

students = new ArrayList<StudentVO>();

StudentVO student1 = new StudentVO("Robert",0);

StudentVO student2 = new StudentVO("John",1);

students.add(student1);

students.add(student2);

}

public void deleteStudent(StudentVO student) {

students.remove(student.getRollNo());

System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");

}

//retrive list of students from the database

public List<StudentVO> getAllStudents() {

return students;

}

public StudentVO getStudent(int rollNo) {

return students.get(rollNo);

}

public void updateStudent(StudentVO student) {

students.get(student.getRollNo()).setName(student.getName());

System.out.println("Student: Roll No " + student.getRollNo() +", updated in the database");

}

}

## Step 3

Use the *StudentBO* to demonstrate Transfer Object Design Pattern.

*TransferObjectPatternDemo.java*

public class TransferObjectPatternDemo {

public static void main(String[] args) {

StudentBO studentBusinessObject = new StudentBO();

//print all students

for (StudentVO student : studentBusinessObject.getAllStudents()) {

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

//update student

StudentVO student = studentBusinessObject.getAllStudents().get(0);

student.setName("Michael");

studentBusinessObject.updateStudent(student);

//get the student

student = studentBusinessObject.getStudent(0);

System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");

}

}

## Step 4

Verify the output.

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]

**SOLID**

## Overview

|  |  |
| --- | --- |
| **Initial** | **Concept** |
| **S** | [Single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)[[4]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-4)  a [class](https://en.wikipedia.org/wiki/Class_(computer_science)) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class) |
| **O** | [Open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle)[[5]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-5)  “software entities … should be open for extension, but closed for modification.” |
| **L** | [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle)[[6]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-6)  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also [design by contract](https://en.wikipedia.org/wiki/Design_by_contract). |
| **I** | [Interface segregation principle](https://en.wikipedia.org/wiki/Interface_segregation_principle)[[7]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-7)  “many client-specific interfaces are better than one general-purpose interface.”[[8]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-martin-design-principles-8) |
| **D** | [Dependency inversion principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle)[[9]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-9)  one should “depend upon abstractions, [not] concretions |

Try [OneTrueError](https://onetrueerror.com/) NOW - Automated exception handling. Know when, where and why exceptions happen!

[jgauffin's coding den](http://blog.gauffin.org/)

Thoughts and ideas about programming in C#/.NET

# SOLID principles with real world examples

The following article aims to explain the five SOLID principles with real world examples. The SOLID principles are five programming principles which is considered to be the foundation of every well designed application. Following the principles will most likely lead to applications which are is easy to extend and maintain. That’s possible since you got small well defined classes with clear contracts.

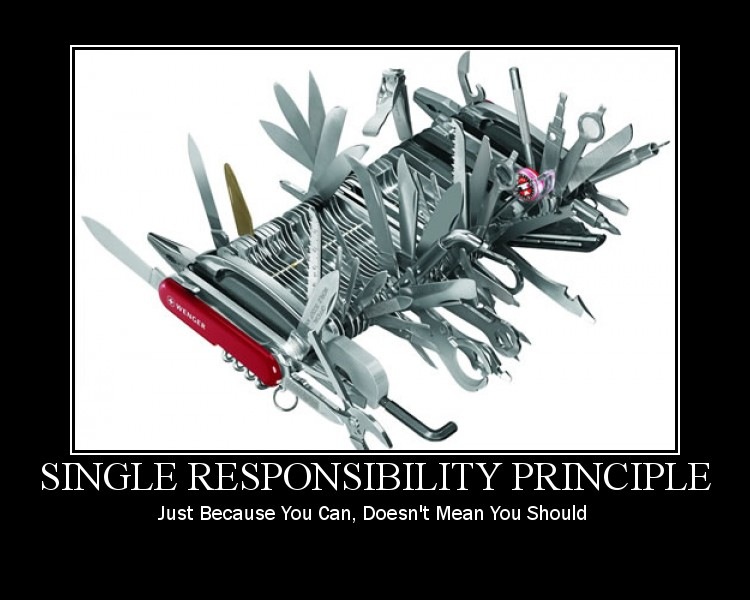


All motivator images are created by [Derick Baley](http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/).

Let’s get started with the principles.

# Single Responsibility Principle

Single responsibility states that every class should only have one reason to change. A typical example is an user management class. When you for instance create a new user you’ll most likely send an welcome email. That’s two reasons to change: To do something with the account management and to change the emailing procedure. A better way would be to generate some kind of event from the account management class which is subscribed by a UserEmailService that does the actual email handling.



The most effective way to break applications it to create [GOD](http://en.wikipedia.org/wiki/God_object) classes. That is classes that keeps track of a lot of information and have several responsibilities. One code change will most likely affect other parts of the class and therefore indirectly all other classes that uses it. That in turn leads to an even bigger maintenance mess since no one dares to do any changes other than adding new functionality to it.

Making sure that a class has a single responsibility makes it per default also easier to see what it does and how you can extend/improve it.

Classes that are hard to unit test are often breaking SRP.

## External links

\* [Wikipedia](http://en.wikipedia.org/wiki/Single_responsibility_principle)  
\* [Example & tips on how to follow the principle (earlier post by me)](http://blog.gauffin.org/2011/07/single-responsibility-prinicple/)

# Open/Closed principle

Open/Closed principle says that a class should be open for extension but closed for modification. Which means that you can add new features through inheritance but should not change the existing classes (other than bug fixes).



The reason is that if you modify a class, you’ll likely break the API/Contract of the class which means that the classes that depend on it might fail when you do so. If you instead inherit the class to add new features, the base contract is untouched and it’s unlikely that dependent classes will fail.

## Example violation

Here is a real world parser method (from a [SO question](http://stackoverflow.com/questions/5416500/understanding-the-open-closed-principle) which I’ve answered):

1. public void Parse()
2. {
3. StringReader reader = new StringReader(scriptTextToProcess);
4. StringBuilder scope = new StringBuilder();
5. string line = reader.ReadLine();
6. while (line != null)
7. {
8. switch (line[0])
9. {
10. case '$':
11. // Process the entire "line" as a variable,
12. // i.e. add it to a collection of KeyValuePair.
13. AddToVariables(line);
14. break;
15. case '!':
16. // Depending of what comes after the '!' character,
17. // process the entire "scope" and/or the command in "line".
18. if (line == "!execute")
19. ExecuteScope(scope);
20. else if (line.StartsWith("!custom\_command"))
21. RunCustomCommand(line, scope);
22. else if (line == "!single\_line\_directive")
23. ProcessDirective(line);
24. scope = new StringBuilder();
25. break;
26. default:
27. // No processing directive, i.e. add the "line"
28. // to the current scope.
29. scope.Append(line);
30. break;
31. }
32. line = reader.ReadLine();
33. }
34. }

It works great. But the method have to be changed each time we want to add support for a new directive. It’s therefore not closed for modification.

## Solution

Lets create an interface which is used for each handler (for '$' and '!' in the example above):

1. public interface IMyHandler
2. {
3. void Process(IProcessContext context, string line);
4. }

Notice that we include a context object. This is quite important. If we create a new parser called SuperCoolParser in the future we can let it create and pass a SuperAwsomeContext to all handlers. New handlers which supports that context can use it while others stick with the basic implementation.

We comply with Liskovs Substitution Principle and doesn’t have to change the IMyHandler.Process signature (and therefore keeping it closed for modification) when we add new features later on.

The parser itself is implemented as:

1. public class Parser
2. {
3. private Dictionary<char, IMyHandler> \_handlers = new Dictionary<char, IMyHandler>();
4. private IMyHandler \_defaultHandler;
5. public void Add(char controlCharacter, IMyHandler handler)
6. {
7. \_handlers.Add(controlCharacter, handler);
8. }
9. private void Parse(TextReader reader)
10. {
11. StringBuilder scope = new StringBuilder();
12. IProcessContext context = null; // create your context here.
13. string line = reader.ReadLine();
14. while (line != null)
15. {
16. IMyHandler handler = null;
17. if (!\_handlers.TryGetValue(line[0], out handler))
18. handler = \_defaultHandler;
19. handler.Process(context, line);
20. line = reader.ReadLine();
21. }
22. }
23. }

If you go back and look at the ! handling you’ll see a lot of if statements. That method likely have to be changed to add support for more features. Hence it do also violate the principle. Let’s refactor again.

1. public interface ICommandHandler
2. {
3. void Handle(ICommandContext context, string commandName, string[] arguments);
4. }
5. public class CommandService : IMyHandler
6. {
7. public void Add(string commandName, ICommandHandler handler)
8. {
9. }
10. public void Handle(IProcessContext context, string line)
11. {
12. // first word on the line is the command, all other words are arguments.
13. // split the string properly
14. // then find the corrext command handler and invoke it.
15. // take the result and add it to the <code>IProcessContext</code>
16. }
17. }

## External links

\* [Wikipedia](http://en.wikipedia.org/wiki/Open/closed_principle)

# Liskovs Substitution Principle

Liskovs Substitution Principle states that any method that takes class X as a parameter must be able to work with any subclasses of X.

The principle makes sure that every class follows the contract defined by its parent class. If the class Car has a method called Break it’s vital that all subclasses breaks when the Break method is invoked. Imagine the suprise if Break() in a Ferrari only works if the switch ChickenMode is activated.



## Violation

Let’s use the motivator image as inspiration and define the following classes:

1. public interface IDuck
2. {
3. void Swim();
4. }
5. public class Duck : IDuck
6. {
7. public void Swim()
8. {
9. //do something to swim
10. }
11. }
12. public class ElectricDuck : IDuck
13. {
14. public void Swim()
15. {
16. if (!IsTurnedOn)
17. return;
18. //swim logic
19. }
20. }

And the calling code:

1. void MakeDuckSwim(IDuck duck)
2. {
3. duck.Swim();
4. }

As you can see, there are two examples of ducks. One regular duck and one electric duck.

The electric duck can only swim if it’s turned on.The MakeDuckSwim method will not work if a duck is electric and not turned on.

This example is incorrect. I mistakenly interpreted LSP as a functional contract, but it’s not. The method will violate LSP if it for instance throws an exception that the base class do not. Read more in my [StackOverflow answer](http://stackoverflow.com/a/4428800/70386).

## Solution

You can of course solve it by doing something like this (in the method that uses the ducks)

1. void MakeDuckSwim(IDuck duck)
2. {
3. if (duck is ElectricDuck)
4. ((ElectricDuck)duck).TurnOn();
5. duck.Swim();
6. }

But that would break Open/Closed principle and has to be implemented everywhere that the ducks are used (and therefore still generate instable code).

The proper solution would be to automatically turn on the duck in the Swim method and by doing so make the electric duck behave exactly as defined by the IDuck interface.

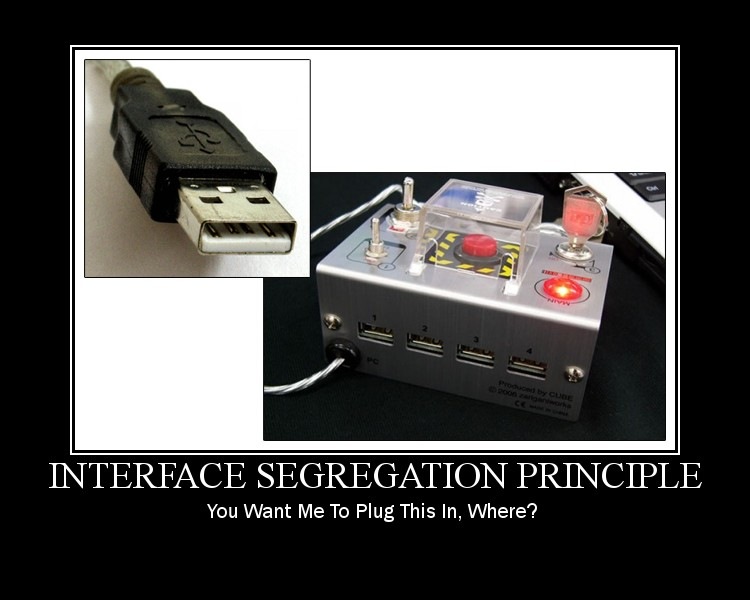
1. public class ElectricDuck : IDuck
2. {
3. public void Swim()
4. {
5. if (!IsTurnedOn)
6. TurnOnDuck();
7. //swim logic
8. }
9. }

## External links

\* [Wikipedia](http://en.wikipedia.org/wiki/Liskov_substitution_principle)

# Interface Segregation Principle

ISP states that interfaces that have become “fat” (like god classes) should be split into several interfaces. Large interfaces makes it harder to extend smaller parts of the system.



There is nothing that says that there should be a one-to-one mapping between classes and interfaces. It’s in fact much better if you can create several smaller interfaces instead (depends on the class though).

## Violation

The MembershipProvider in ASP.NET is a classical example of a violation. [MSDN](http://msdn.microsoft.com/en-us/library/f1kyba5e.aspx/) contains a large article (which 4 out of 34 have found useful) which contains a long and detailed instruction on how to properly implement the class.

## Solution

The provider could have been divided in several parts:

\* MembershipProvider – A facade to the below interfaces  
\* IAccountRepository – Used to fetch/load accounts  
\* IPasswordValidator – Checks that the password is valid (according to business rules)  
\* IPasswordStrategy – How to store PW (hash it, use encryption or just store it in plain text)

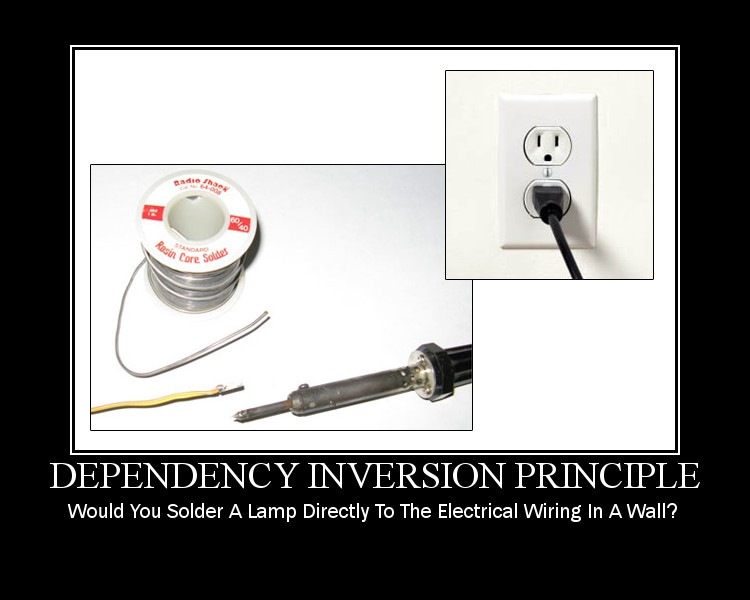
Now you only have to implement a small part if you need to customize the provider (for instance IAccountRepository if you are using a custom data source). There’s a saying: “Favor composition over inheritance” which the membership providers illustrates well. The original solution used inheritance while mine used composition.

## External Links

\* [Wikipedia](http://en.wikipedia.org/wiki/Interface_segregation_principle)

# Dependency Inversion Principle

The principle which is easiest to understand. DIP states that you should let the caller create the dependencies instead of letting the class itself create the dependencies. Hence inverting the dependency control (from letting the class control them to letting the caller control them).



## Before

1. public class Volvo
2. {
3. B20 \_engine;
4. public Volvo()
5. {
6. \_engine = new B20();
7. }
8. }

## After

1. public class Volvo
2. {
3. IEngine \_engine;
4. public Volvo(IEngine engine)
5. {
6. if (engine == null) throw new ArgumentNullException("engine");
7. \_engine = engine;
8. }
9. }

Which makes it a lot more fun since we now can do the following:

1. var myVolvo = new Volvo(new BigBadV12());

(Nice real world example, huh? ;))

## Update

I messed up a bit. Dependency Inversion states that you should depend upon abstractions and that lower layers should not be aware of higher layers. I’ll get back to that. Dependency Injection is when you inject dependencies into the constructors instead of creating them in the class. Inversion Of Control is that the container controls your objects and their lifetime.

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.  
B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS

The original principle targets modules while I also like to apply it at class level too. It makes the principle easier to apply (so the text below is how I apply the principle).

Depend on abstractions simply means that you should depend on as generic class/interface as possible. For instance IUserRepository instead of DbUserRepository or HttpRequestBase instead of HttpRequest. The purpose is that your code should be as flexible as possible. The more abstract the dependencies are, the easier it is to refactor them or create new implementations.

Depending on abstractions also make your code less likely to change if the dependency change.

## External links

\* [Wikipedia](http://en.wikipedia.org/wiki/Dependency_inversion_principle)  
\* [Introduction](http://joelabrahamsson.com/entry/inversion-of-control-introduction-with-examples-in-dotnet)

# Summary

Feel free to leave a comment if something is unclear. It will not just help you, but all future readers.