Core Java Design Patterns

# **Creational Patterns (5)**

**Singleton**

**Factory**

**Abstract Factory**

**Builder**

**Prototype**

The **Factory** **Builder** created **Singleton** **Prototype** from **Abstract Factory**.

# **Structural Patterns (7)**

**Adapter**

**Bridge**

**Composite**

**Decorator**

**Facade**

**Flyweight**

**Proxy**

The **Facade** structured the **Decorated** **Bridge** **Adapter** for **Composite** **Proxy** on the **Flyweight**.

# **Behavioral Patterns (11)**

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template

Visitor

The **Interpreter** gave the **Memento** of **State** **Strategy** **Template** for **Chain of Responsibility** **Command**

to

**Visitor, Mediator and Observer ( VOM – Visitor Observer Mediator )**

for

**Iteration**

Singleton Design Pattern

GOF : **Ensure a class only has one instance, and provide a global point of access to it.**

Here we will discuss Singleton Design Concept on the following aspects.

**1. Normal way of creating Singleton Design Class**

**2. Singleton Class in case Reflection and Introspection**

**3. Singleton Class using Security Manager**

**4. Singleton Class in case of Serialization**

**5. Singleton Class in case of Multi threading**

**6. Singleton Class using Double Checked Locking mechanism**

**7. Singleton Class using Initialization on Demand mechanism(Bill Pugh's Solution)**

**8. Singleton Class using Java's Enum Concept suggested by Joshua Bloch**

**9. Cracking of Singleton using a nasty approach**

A typical fragile Singleton design pattern is given below.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

To use and implement we have to write the following code.

SingletonType1 instance = SingletonType1.getInstance();

It is fine and seems to be correct. If you write the above code 10 times or more you will get the same instance. If you do not override the hashcode() and toString() method , you will get the same evertime when you invoke. Let us consider another test.Create the instances of the above class by invoking "SingletonType1.getInstance()" the code and put all the instances in a Set. As you know that Set does not allow duplicates. So at the end if you are getting the size of the set 1, then it is the correct implementation. But the above design can be broken and we can create multiple instances of the Singleton design pattern using reflection.

**Note: You can use java's Reflection or Introspection utility to invoke the instance of Singleton design class if there is no adequete protection .**

Let us see the code below.

package com.ddlab.rnd.patterns;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class TestSingletonType1 {

public void createMultiInstances()

{

System.out.println("\n\*\*\*\*\* MULTIPLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\*\*\*\n");

/\*

\* Using Reflection you can break singleton

\*/

try {

**Class clazz = Class.forName("com.ddlab.rnd.patterns.SingletonType1");**

**Constructor constructor = clazz.getDeclaredConstructors()[0];**

**constructor.setAccessible(true);**

**SingletonType1 instance1 = (SingletonType1)constructor.newInstance(null);**

SingletonType1 instance2 = (SingletonType1)constructor.newInstance(null);

SingletonType1 instance3 = (SingletonType1)constructor.newInstance(null);

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "MULTI INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "---------------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

catch (Exception e) {

e.printStackTrace();

}

}

public void createMultiInstances1() {

System.out.println("\n\*\*\*\*\*\*\*\* MULTIPLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\n");

/\*

\* Using Reflection you can break singleton

\*/

try

{

Class clazz = Class.forName("com.ddlab.rnd.patterns.SingletonType1");

Method method = clazz.getDeclaredMethods()[0];

Field field = clazz.getDeclaredFields()[0];

field.setAccessible(true);

SingletonType1 instance1 = (SingletonType1)method.invoke(clazz, null);

field.set(clazz, null);

SingletonType1 instance2 = (SingletonType1)method.invoke(clazz, null);

field.set(clazz, null);

SingletonType1 instance3 = (SingletonType1)method.invoke(clazz, null);

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "MULTI INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "---------------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

catch (Exception e) {

e.printStackTrace();

}

}

public void createInstances()

{

System.out.println("\n\*\*\*\*\*\*\*\* SINGLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\*\*\n");

SingletonType1 instance1 = SingletonType1.getInstance();

SingletonType1 instance2 = SingletonType1.getInstance();

SingletonType1 instance3 = SingletonType1.getInstance();

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "----------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

public static void main(String[] args)

{

new TestSingletonType1().createInstances();

new TestSingletonType1().createMultiInstances();

new TestSingletonType1().createMultiInstances1();

}

}

If you run the above program you will be able to see the many instances of the defined singleton class.

However we got to know to know the private constructor approach of Singleton can be broken using reflection. In the above case we are able to create the instance of a class having private constructor and also we are able to access the private field. We are also able to invoke the getInstance() method by accessing the private field.

**If you want to use Java's Introspection utility, the code is given below.**

public void doIt() {

try {

MethodDescriptor[] mds = Introspector.getBeanInfo(SingletonType2.class).getMethodDescriptors();

for( MethodDescriptor methodDescs : mds )

{

Method method = methodDescs.getMethod();

if( method.getName().equals("getInstance"))

{

SingletonType2 instance1 = (SingletonType2)method.invoke(SingletonType2.class, null);

System.out.println("First Instance :::"+instance1);

Field underLyingField = method.getDeclaringClass().getDeclaredField("instance");

underLyingField.setAccessible(true);

underLyingField.set(Introspector.getBeanInfo(SingletonType2.class), null);

SingletonType2 instance2 = (SingletonType2)method.invoke(SingletonType2.class, null);

System.out.println("Second Instance :::"+instance2);

}

}

}

catch (Exception e)

{

e.printStackTrace();

}

}

To gurad against the reflection and Introspection attack, we can write a protective code not the robust one. Now let us see how to write a better code so that other developers will not be able to break using reflection.

package com.ddlab.rnd.patterns;

import java.lang.reflect.ReflectPermission;

import java.security.Permission;

public class SingletonType2

{

static

{

getInstance();

}

private static SingletonType2 instance = null;

**private SingletonType2()**

**{**

**super();**

**//Add the following piece of code so that it can not be invoked using relection**

**System.setSecurityManager(new SecurityManager() {**

**@Override**

**public void checkPermission(Permission perm)**

**{**

**if (perm instanceof ReflectPermission )**

**{**

**System.out.println("\nYes I will not allow you to create the instance using Reflection...\n");**

**throw new SecurityException();**

**}**

**else**

**{**

**//Do nothing**

**}**

**}**

**});**

**}**

public static SingletonType2 getInstance() {

if( instance == null )

instance = new SingletonType2();

return instance;

}

}

Now it is true, your reflection attack will not impact the above code. You will get Exception if you use reflection to create another instance. The following piece of code will help you against the reflection attack.

**System.setSecurityManager(new SecurityManager() {**

**@Override**

**public void checkPermission(Permission perm) {**

**if (perm instanceof ReflectPermission ) {**

**System.out.println("\nYes I will not allow you to create the instance using Reflection...\n");**

**throw new SecurityException();**

**}**

**else {**

**//Do nothing**

**}**

**}**

**});**

However you can use java's introspection utility to create the instances of the Singleton Class, but you will always get the same instance.

**What about serialization ?**

We can not serialize the class SingletonType2 as it does not implement Serializable interface and also we do not allow reflection. However we can not serialize a class which does not implement Serilizable interface. However sometimes it is required to persist the Singleton object for a day. In this case we have to implement Serilizable interface in singleton class.

Let us see the Singleton class with Serilizable interface.

package com.ddlab.rnd.patterns;

import java.io.Serializable;

public class SingletonType11 implements Serializable

{

private static final long serialVersionUID = -4137189065490862968L;

private static SingletonType11 instance = null;

private SingletonType11()

{

super();

}

public static SingletonType11 getInstance()

{

if( instance == null )

instance = new SingletonType11();

return instance;

}

}

Let us see how we can again break the concept of Singleton through object serilization.

Let use write a small class like this.

package com.ddlab.rnd.patterns;

import java.io.Serializable;

public class BreakSingleton implements Serializable

{

private static final long serialVersionUID = 5904306999023481976L;

private SingletonType11 instance2 = SingletonType11.getInstance();

public SingletonType11 getInstance2() {

return instance2;

}

public void setInstance1(SingletonType11 instance2) {

this.instance2 = instance2;

}

}

Let us see the test harness class for the above.

package com.ddlab.rnd.patterns;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.InputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.OutputStream;

public class TestBreakSingleton

{

public static void main(String[] args) throws Exception

{

BreakSingleton bs = new BreakSingleton();

OutputStream out = new FileOutputStream("data/a.ser");

ObjectOutputStream oout = new ObjectOutputStream(out);

oout.writeObject(bs);

oout.flush();

oout.close();

out.flush();

out.close();

InputStream in = new FileInputStream("data/a.ser");

ObjectInputStream oin = new ObjectInputStream(in);

BreakSingleton bs1 = (BreakSingleton)oin.readObject();

oin.close();

in.close();

System.out.println("Instance from Serialization :::"+bs1.getInstance2());

System.out.println("Normal Instance :::"+SingletonType11.getInstance());

InputStream in1 = new FileInputStream("data/a.ser");

ObjectInputStream oin1 = new ObjectInputStream(in1);

BreakSingleton bs2 = (BreakSingleton)oin1.readObject();

oin1.close();

in1.close();

System.out.println("Another Instance from Serialization :::"+bs2.getInstance2());

}

}

If you run the above program, you will get following type of output.

Instance from Serialization :::com.ddlab.rnd.patterns.SingletonType11@2586db54

Normal Instance :::com.ddlab.rnd.patterns.SingletonType11@12276af2

Another Instance from Serialization :::com.ddlab.rnd.patterns.SingletonType11@38a97b0b

So now you got three different instances of a Singleton class. Now let us see the modified singleton java class so that we will be able to serialize the object and at any point of time we will get the consistent singleton class.

package com.ddlab.rnd.patterns;

import java.io.ObjectStreamException;

import java.io.Serializable;

public class SingletonType11 implements Serializable

{

private static final long serialVersionUID = -4137189065490862968L;

private static SingletonType11 instance = null;

private SingletonType11()

{

super();

}

public static SingletonType11 getInstance()

{

if( instance == null )

instance = new SingletonType11();

return instance;

}

private Object readResolve() throws ObjectStreamException

{

return instance;

}

private Object writeReplace() throws ObjectStreamException

{

return instance;

}

}

However still we can create multiple instance using Reflection and we can not prevent reflection as we want to serialize the object. We are giving an opertunity to hacker to create multiple instance.

Let us see the use of thread in case of Singleton class.

Let us consider the same class,

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

The above approach is called lazy-initialization. In case of multi threading we can get multiple instances. Let see the code below.

package com.ddlab.rnd.patterns;

import java.util.Collections;

import java.util.HashSet;

import java.util.Set;

class Thread1 extends Thread

{

@Override

public void run()

{

SingletonType1 instance = SingletonType1.getInstance();

// System.out.println("In Thread 1 - Singleton Instance ---->"+instance);

TestSingletonType1\_Thread.singletonSet.add(instance);

}

}

class Thread2 extends Thread

{

@Override

public void run()

{

SingletonType1 instance = SingletonType1.getInstance();

// System.out.println("In Thread 2 - Singleton Instance ---->"+instance);

TestSingletonType1\_Thread.singletonSet.add(instance);

}

}

public class TestSingletonType1\_Thread

{

private static Set<SingletonType1> singletonSet1 = new HashSet<SingletonType1>();

public static Set<SingletonType1> singletonSet = Collections.synchronizedSet(singletonSet1);

public static void main(String[] args)

{

//Singleton concept is broken here

for( int i = 0 ; i < 100 ; i++ )

{

new Thread1().start();

new Thread2().start();

if( singletonSet.size() > 1 )

break;

else

continue;

}

System.out.println(singletonSet);

}

}

If you run the above program many times, you will get the different instances of the Singleton class.

Upon running the program, you may get a similar type result like this.

[com.ddlab.rnd.patterns.SingletonType1@60723d7c, com.ddlab.rnd.patterns.SingletonType1@6d9efb05, com.ddlab.rnd.patterns.SingletonType1@8dd20f6]

So what to do ? Can we declare volatile variable, let see now.

Let us have the modified program.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

Upon running the program , you may get like this.

[com.ddlab.rnd.patterns.SingletonType1@3f0ef90c, com.ddlab.rnd.patterns.SingletonType1@2e471e30]

However use of volatile does not serve our purpose here.

Still the same problem, can we use synchronized method, yes we can do it.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static synchronized SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

But there will be a performance issue. However, when you analyze it you realize that synchronization is required only

for the first invocation of the method. Subsequent invocations do not require synchronization.

In order to increase the efficiency of the above program, let use modify the above program in a different manner.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if (instance == null)

{

synchronized(SingletonType1.class) {

instance = new SingletonType1();

}

}

return instance;

}

}

The above program looks ok. But still there is a very big problem.

Two threads can get inside of the if statement concurrently when instance is null.

Then, one thread enters the synchronized block to initialize instance,

while the other is blocked. When the first thread exits the synchronized block,

the waiting thread enters and creates another Singleton object.

Note that when the second thread enters the synchronized block, it does not check to see if instance is non-null.

**Double-Checked Locking Idiom in Singleton Design Pattern**

Now let us consider another concept called "double-checked locking" which seems to be famous for a set of developers. Many developers apply in many cases and think to be most robust form of singleton. In software engineering, double-checked locking (also known as "double-checked locking optimization"[1]) is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") without actually acquiring the lock. Only if the locking criterion check indicates that locking is required does the actual locking logic proceed.

The structure is given below.

public static SingletonType1 getInstance()

{

if (instance == null)

{

synchronized(SingletonType1.class) // Mark - 1

{

if (instance == null) // Mark - 2

instance = new SingletonType1(); // Mark - 3

}

}

return instance;

}

The theory behind double-checked locking is that the second check at // Mark - 2 makes it impossible for two different Singleton objects to be created. It may be true for a single threaded application. What about fine grained multi-threaded application ?

Thread 1 enters the getInstance() method.

Thread 1 enters the synchronized block at // Mark - 1 because instance is null.

Thread 1 is preempted by thread 2

Thread 2 enters the getInstance() method.

Thread 2 attempts to acquire the lock at // Mark - 1 because instance is still null. However, because thread 1 holds the lock, thread 2 blocks at // Mark - 1.

Thread 2 is preempted by thread 1.

Thread 1 executes and because instance is still null at // Mark - 2, creates a Singleton object and assigns its reference to instance.

Thread 1 exits the synchronized block and returns instance from the getInstance() method.

Thread 1 is preempted by thread 2.

Thread 2 acquires the lock at // Mark - 1 and checks to see if instance is null.

Because instance is non-null, a second Singleton object is not created and the one created by thread 1 is returned. The theory behind double-checked locking is perfect. Unfortunately, reality is entirely different. The problem with double-checked locking is that there is no guarantee it will work on single or multi-processor machines. The issue of the failure of double-checked locking is not due to implementation bugs in JVMs but to the current Java platform memory model. The memory model allows what is known as "out-of-order writes" and is a prime reason why this idiom fails. However the concept "out-of-order writes" is beyond the scope of our discussion.

The bottom line is that double-checked locking, in whatever form, should not be used because you cannot guarantee that it will work on any JVM implementation. As we have seen that although "double-checked-locking" may work but it may fail unexpectedly.

**The solution of Bill Pugh**

University of Maryland Computer Science researcher Bill Pugh has written about the code issues underlying the Singleton pattern when implemented in Java.[9] Pugh's efforts on the "Double-checked locking" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to implement Singletons in Java. The technique known as the initialization on demand holder idiom, is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines. The nested class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that getInstance() is called. Thus, this solution is thread-safe without requiring special language constructs (i.e. volatile or synchronized).

**public class Singleton**

**{**

**// Private constructor prevents instantiation from other classes**

**private Singleton() { }**

**/\*\***

**\* SingletonHolder is loaded on the first execution of Singleton.getInstance()**

**\* or the first access to SingletonHolder.INSTANCE, not before.**

**\*/**

**private static class SingletonHolder {**

**public static final Singleton INSTANCE = new Singleton();**

**}**

**public static Singleton getInstance() {**

**return SingletonHolder.INSTANCE;**

**}**

**}**

The above is called "Initialization-on-demand holder idiom".

Let us understand the concept.

**public class Something {**

**private Something() {**

**}**

**private static class LazyHolder {**

**public static final Something INSTANCE = new Something();**

**}**

**public static Something getInstance() {**

**return LazyHolder.INSTANCE;**

**}**

**}**

**How it works**

The implementation relies on the well-specified initialization phase of execution within the Java Virtual Machine (JVM); see section 12.4 of Java Language Specification (JLS) for details.

When the class Something is loaded by the JVM, the class goes through initialization. Since the class does not have any static variables to initialize, the initialization completes trivially. The static class definition LazyHolder within it is not initialized until the JVM determines that LazyHolder must be executed. The static class LazyHolder is only executed when the static method getInstance is invoked on the class Something, and the first time this happens the JVM will load and initialize the LazyHolder class. The initialization of the LazyHolder class results in static variable INSTANCE being initialized by executing the (private) constructor for the outer class Something. Since the class initialization phase is guaranteed by the JLS to be serial, i.e., non-concurrent, no further synchronization is required in the static getInstance method during loading and initialization. And since the initialization phase writes the static variable INSTANCE in a serial operation, all subsequent concurrent invocations of the getInstance will return the same correctly initialized INSTANCE without incurring any additional synchronization overhead. However using the concept of "Initialization-on-demand holder idiom" pattern we can achieve a thread-safe singleton construct. Again the question arises, can we break reflectively. Yes we can break the above concept using java reflection mechnism which I have already mentioned. Now question comes , is there any other approach to build a proper singleton design approach. Yes there is another one as suggested by

Joshua Bloch(Chief Technical Architect, Google Innovation Lab and Author of famous Book Effective Java).

**package com.ddlab.rnd.patterns;**

**public enum SingletonType3**

**{**

**INSTANCE;**

**public void doSomething(String arg)**

**{**

**//... perform operation here ...**

**}**

**}**

This is the only robust approach of creating a singleton class which is serializable and completely thread-safe as enums by default thread-safe.

What about reflection, using our above normal approach of reflection, you can not break singleton object as it has no constructor. **What about serialization**, you will be able to serialize it but you will get the same instance every time. So finally we have to imbibe this modern

approach of creation of Singleton design class. However many developers do not know about it.

Still we can not break the above approach of singleton, but we can crack the above by writing a code to create more than one instance.

The code is give below do not use the below code for your commercial product. It is a nasty approach of breaking singleton.

package com.ddlab.rnd.patterns;

import java.lang.reflect.Constructor;

import java.lang.reflect.Method;

import java.util.HashSet;

import java.util.Set;

import sun.reflect.ConstructorAccessor;

public class CrackEnumSingleton

{

public static void main(String[] args)

{

Set set = new HashSet();

try

{

SingletonType3 firstInstance = SingletonType3.INSTANCE;

System.out.println(firstInstance.getClass() + " " + firstInstance + " = " + System.identityHashCode(firstInstance));

set.add(firstInstance);

Constructor constructor = SingletonType3.class.getDeclaredConstructors()[0];

Method acquire = constructor.getClass().getDeclaredMethod("acquireConstructorAccessor");//"acquireConstructorAccessor" fields for cracking

acquire.setAccessible(true);

acquire.invoke(constructor);

Method get = constructor.getClass().getDeclaredMethod("getConstructorAccessor");//"getConstructorAccessor" fields for cracking

get.setAccessible(true);

ConstructorAccessor invoke = (ConstructorAccessor) get.invoke(constructor);

Object secondInstance = invoke.newInstance(new Object[] {null,1});

System.out.println(secondInstance.getClass() + " " + secondInstance + " = " + System.identityHashCode(secondInstance));

set.add(secondInstance);

System.out.println("Total No of Singletons :::"+set.size());

}

catch (Exception e)

{

e.printStackTrace();

}

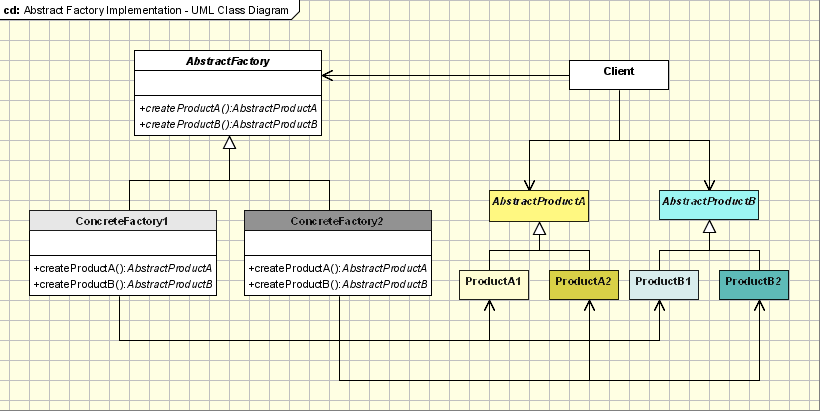
}

}

Abstract Factory

GOF : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

# **Class Diagram**



# **Java Code**

**public** **abstract** **class** AbstractFactory {

**abstract** AbstractProductA createProductA();

//Create another product

// abstract AbstractProductB createProductB();

}

**abstract** **class** AbstractProductA {

**public** **abstract** **void** operation1();

}

**class** ConcreteFactory1 **extends** AbstractFactory {

@Override

AbstractProductA createProductA() {

**return** **new** ProductA1();

}

//Create for another Product

}

**public** **class** ProductA1 **extends** AbstractProductA {

@Override

**public** **void** operation1() {

System.***out***.println("Operation1 executed ...");

}

}

**public** **class** FactoryMaker {

**private** **static** AbstractFactory *absFactory* = **null**;

**public** **static** AbstractFactory getFactory(String choice) {

**if** (choice.equals("a")) {

*absFactory* = **new** ConcreteFactory1();

} **else** {

// return another ConcreteFactory

}

**return** *absFactory*;

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

AbstractFactory absFact = FactoryMaker.*getFactory*("a");

AbstractProductA prodA = absFact.createProductA();

prodA.operation1();

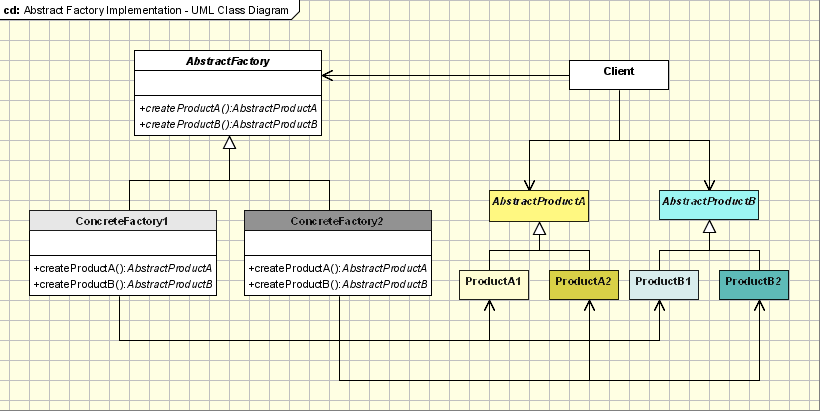
}

}

Abstract Factory

GOF : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

# **Class Diagram**



# **Java Code**

abstract class AbstractProductB{

//public abstract void operationB1();

//public abstract void operationB2();

}

abstract class AbstractProductA {

public abstract void operationA1();

public abstract void operationA2();

}

class ProductA2 extends AbstractProductA {

ProductA2(String arg) {

System.out.println("Hello "+arg);

} // Implement the code here

public void operationA1() { };

public void operationA2() { };

}

class ProductA1 extends AbstractProductA {

ProductA1(String arg){

System.out.println("Hello "+arg);

} // Implement the code here

public void operationA1() { };

public void operationA2() { };

}

abstract class AbstractFactory{

abstract AbstractProductA createProductA();

abstract AbstractProductB createProductB();

}

class ConcreteFactory2 extends AbstractFactory {

AbstractProductA createProductA() {

return new ProductA2("ProductA2");

}

AbstractProductB createProductB(){

return new ProductB2("ProductB2");

}

}

class ConcreteFactory1 extends AbstractFactory {

AbstractProductA createProductA() {

return new ProductA1("ProductA1");

}

AbstractProductB createProductB() {

return new ProductB1("ProductB1");

}

}

class ProductB2 extends AbstractProductB {

ProductB2(String arg) {

System.out.println("Hello "+arg);

} // Implement the code here

}

class ProductB1 extends AbstractProductB {

ProductB1(String arg) {

System.out.println("Hello "+arg);

} // Implement the code here

}

//Factory creator - an indirect way of instantiating the factories

class FactoryMaker{

private static AbstractFactory pf=null;

static AbstractFactory getFactory(String choice) {

if(choice.equals("a")) {

pf=new ConcreteFactory1();

}else if(choice.equals("b")) {

pf=new ConcreteFactory2();

} return pf;

}

}

// Client

public class Client {

public static void main(String args[]) {

AbstractFactory pf=FactoryMaker.getFactory("a");

AbstractProductA product=pf.createProductA();

//more function calls on product

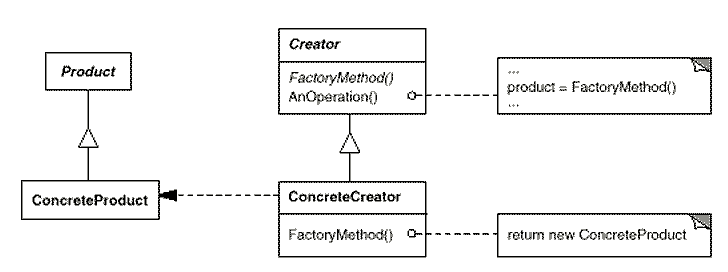
}

}

Factory Pattern

GOF : **Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.**

# **Structure Diagram**



Java Code

**class** ConcreteProductA **implements** Product {

@Override

**public** String getName() {

**return** "Product-A";

}

}

**interface** Product {

String getName();

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

AbstractProductCreator creator = **new** ConcreteCreator();

Product product = creator.getProduct("A");

System.***out***.println(product.getName());

}

}

**class** ConcreteCreator **extends** AbstractProductCreator {

**private** Product product = **null**;

**public** Product getProduct(String type) {

**if** (type.equals("A"))

product = **new** ConcreteProductA();

**else**

product = **new** ConcreteProductA();

**return** product;

}

}

**public** **abstract** **class** AbstractProductCreator {

**public** **abstract** Product getProduct(String type);

}

**class** ConcreteProductB **implements** Product {

@Override

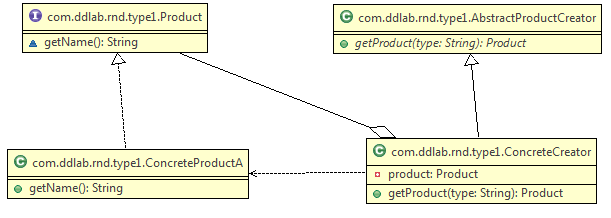
**public** String getName() {

**return** "Product-B";

}

}

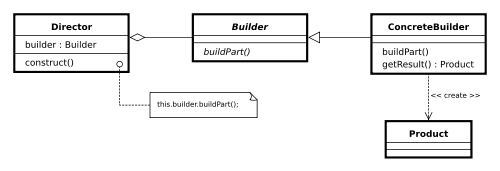
Class Diagram



Builder Design Pattern

GOF : **Separate the construction of a complex object from its representation so that the same construction process can create different representations.**

# **Class Diagram**

  
**Java Code**

## **Vehicle.java**

**package** com.ddlab.rnd.type3;

**public** **class** Vehicle {

**private** String vehicleName;

**private** String engineType;

**private** String vehicleType;

**public** String getVehicleName() {

**return** vehicleName;

}

**public** **void** setVehicleName(String vehicleName) {

**this**.vehicleName = vehicleName;

}

**public** String getEngineType() {

**return** engineType;

}

**public** **void** setEngineType(String engineType) {

**this**.engineType = engineType;

}

**public** String getVehicleType() {

**return** vehicleType;

}

**public** **void** setVehicleType(String vehicleType) {

**this**.vehicleType = vehicleType;

}

@Override

**public** String toString() {

**return** "Vehicle [vehicleName=" + vehicleName + ", " + "engineType=" + engineType + ", " + "vehicleType="

+ vehicleType + "]";

}

}

## **IVehicleBuilder.java**

**package** com.ddlab.rnd.type3;

**public** **interface** IVehicleBuilder {

**void** buildEngine(String engineType);

**void** buildBody(String type);

Vehicle getVehicle();

}

## **CarBuilder.java**

**package** com.ddlab.rnd.type3;

**public** **class** CarBuilder **implements** IVehicleBuilder {

**private** Vehicle vehicle = **new** Vehicle();

@Override

**public** **void** buildBody(String type) {

vehicle.setVehicleType(type);

}

@Override

**public** **void** buildEngine(String engineType) {

vehicle.setEngineType(engineType);

vehicle.setVehicleName("Car");

}

@Override

**public** Vehicle getVehicle() {

**return** vehicle;

}

}

## **CarDirector.java**

**package** com.ddlab.rnd.type3;

**public** **class** CarDirector {

**private** IVehicleBuilder builder;

**public** CarDirector(IVehicleBuilder builder) {

**this**.builder = builder;

}

**public** **void** buildVehicle() {

builder.buildBody("Sedan+Luxury");

builder.buildEngine("TurboJet");

}

**public** Vehicle getVehicle() {

**return** builder.getVehicle();

}

}

## **Test.java**

**package** com.ddlab.rnd.type3;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

CarDirector director = **new** CarDirector( **new** CarBuilder() );

director.buildVehicle();

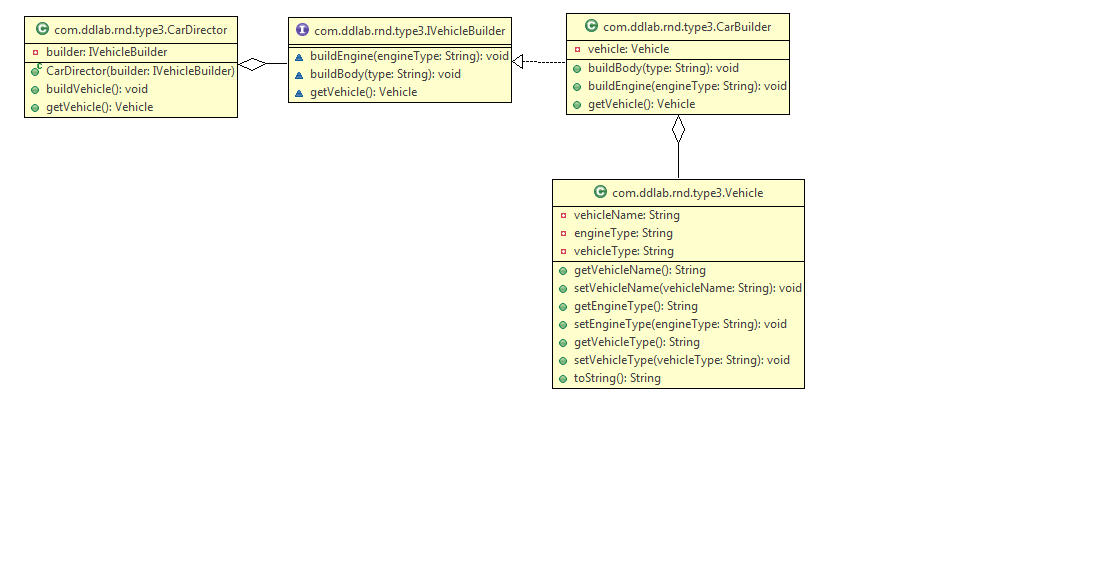
Vehicle vehicle = director.getVehicle();

System.***out***.println(vehicle);

}

}

The UML class diagram is given below.



Builder Design Pattern Example

**IBuilder.java**

public interface IBuilder {  
 void buildBody();  
 Product getProduct();  
}

**ConcreteBuilder.java**

public class ConcreteBuilder implements IBuilder {  
 private Product product = new Product();  
  
 @Override  
 public void buildBody() {  
 System.*out*.println("... body building ...");  
 product.setName("Prod Name 1");  
 }  
  
 @Override  
 public Product getProduct() {  
 return product;  
 }  
}

**Director.java**

public class Director {  
  
 private IBuilder builder = null;  
  
 public Director(IBuilder builder ) {  
 this.builder = builder;  
 }  
  
 public void build() {  
 builder.buildBody();  
 }  
  
 public Product getProduct() {  
 return builder.getProduct();  
 }  
}

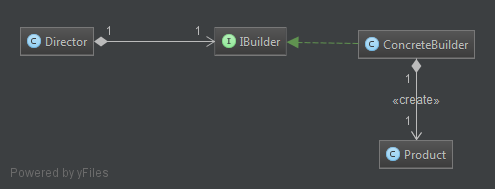
**Product.java**

public class Product {  
 private String name;  
 private String type;  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
}

**Test.java**

public class Test {  
  
 public static void main(String[] args) {  
 Director director = new Director( new ConcreteBuilder());  
 director.build();  
 Product product = director.getProduct();  
 System.*out*.println("Product Name :::"+product.getName());  
  
 }  
}

The generated class diagram is given below.



Builder Pattern using Fluent Interface

Below are some reasons arguing for the use of the pattern and example code in Java, but it is an implementation of the Builder Pattern covered by the Gang of Four in *Design Patterns*. The reasons you would use it in Java are also applicable to other programming languages as well.

As Joshua Bloch states in [Effective Java, 2nd Edition](http://www.amazon.co.uk/Effective-Java-Second-Joshua-Bloch/dp/0321356683):

The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

We've all at some point encountered a class with a list of constructors where each addition adds a new option parameter:

Pizza(int size) { ... }

Pizza(int size, boolean cheese) { ... }

Pizza(int size, boolean cheese, boolean pepperoni) { ... }

Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { ... }

**This is called the Telescoping Constructor Pattern.** The problem with this pattern is that once constructors are 4 or 5 parameters long it becomes **difficult to remember** the required **order of the parameters** as well as what particular constructor you might want in a given situation.

One **alternative** you have to the Telescoping Constructor Pattern is the **JavaBean Pattern** where you call a constructor with the mandatory parameters and then call any optional setters after:

Pizza pizza = new Pizza(12);

pizza.setCheese(true);

pizza.setPepperoni(true);

pizza.setBacon(true);

**The problem here is that because the object is created over several calls it may be in an inconsistent state partway through its construction.** This also requires a lot of extra effort to ensure thread safety.

**The better alternative is to use the Builder Pattern.**

**This results in code that is easy to write and very easy to read and understand.** In this example, the **build method could be modified** to check parameters after they have been copied from the builder to the Pizza object and **throw an IllegalStateException if an invalid parameter value has been supplied.** This pattern is flexible and it is easy to add more parameters to it in the future. It is really only useful if you are going to have more than 4 or 5 parameters for a constructor. That said, it might be worthwhile in the first place **if you suspect you may be adding more parameters in the future.**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Pizza pizza = **new** Pizza.Builder(12)

.cheese(**true**)

.pepperoni(**true**)

.bacon(**true**)

.build();

}

}

**public** **class** Pizza {

**private** **int** size;

**private** **boolean** cheese;

**private** **boolean** pepperoni;

**private** **boolean** bacon;

**public** **static** **class** Builder {

// required

**private** **final** **int** size;

// optional

**private** **boolean** cheese = **false**;

**private** **boolean** pepperoni = **false**;

**private** **boolean** bacon = **false**;

**public** Builder(**int** size) {

**this**.size = size;

}

**public** Builder cheese(**boolean** value) {

cheese = value;

**return** **this**;

}

**public** Builder pepperoni(**boolean** value) {

pepperoni = value;

**return** **this**;

}

**public** Builder bacon(**boolean** value) {

bacon = value;

**return** **this**;

}

**public** Pizza build() {

**return** **new** Pizza(**this**);

}

}

**private** Pizza(Builder builder) {

size = builder.size;

cheese = builder.cheese;

pepperoni = builder.pepperoni;

bacon = builder.bacon;

}

}

In case of Jersey, the Respponse is built like this.

Response.ok(200).entity(“Something”).build();

In case of Hibernate, the query can be formed like,

HQLQuery query = new HibernateQuery(session);

List<Stock> s = query.from(stock, bonus)

.where(stock.someValue.eq(bonus.id))

.list(stock);

One more example is given below.

**package** com.ddlab.rnd.type2;

**public** **class** Car {

**private** **int** quantity;

// Optional Parameters

**private** **boolean** isAVS;

**private** **boolean** isAutomaticGear;

**private** **boolean** isBulletProof;

**public** **static** **class** CarBuilder {

**private** **int** quantity;

**private** **boolean** isAVS;

**private** **boolean** isAutomaticGear;

**private** **boolean** isBulletProof;

**public** CarBuilder(**int** quantity) {

**this**.quantity = quantity;

}

**public** CarBuilder avsSystem(**boolean** flag) {

isAVS = flag;

**return** **this**;

}

**public** CarBuilder automaticGear(**boolean** flag) {

isAutomaticGear = flag;

**return** **this**;

}

**public** CarBuilder bulletProof(**boolean** flag) {

isBulletProof = flag;

**return** **this**;

}

**public** Car build() {

**return** **new** Car(**this**);

}

}

**private** Car(CarBuilder builder) {

isAVS = builder.isAVS;

isAutomaticGear = builder.isAutomaticGear;

isBulletProof = builder.isBulletProof;

}

}

**public** **class** TestCar {

**public** **static** **void** main(String[] args) {

Car car = **new** Car.CarBuilder(1).automaticGear(**true**).bulletProof(**true**).build();

System.***out***.println(car);

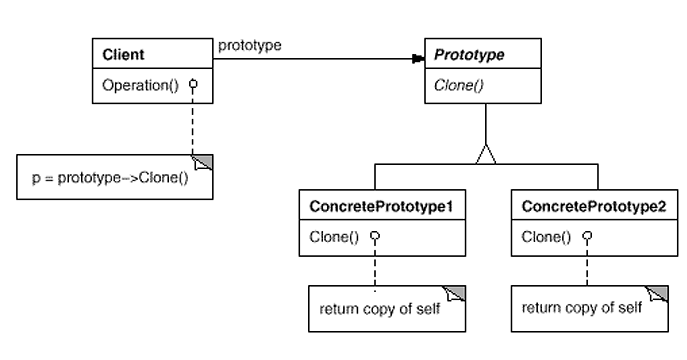
}

}

Prototype Design Pattern

GOF : **Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.**

# **Structure Diagram**



# **Java Code Example**

public interface Prototype {

public abstract Object clone ( );

}

public class ConcretePrototype implements Prototype {

public Object clone() {

return super.clone();

}

}

public class Client {

public static void main( String arg[] ) {

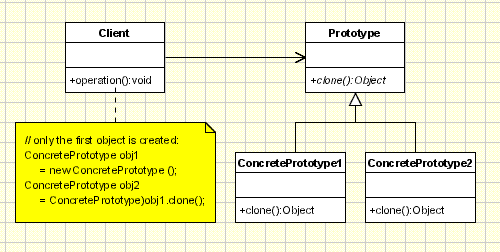
ConcretePrototype obj1= new ConcretePrototype ();

ConcretePrototype obj2 = ConcretePrototype)obj1.clone();

}

}

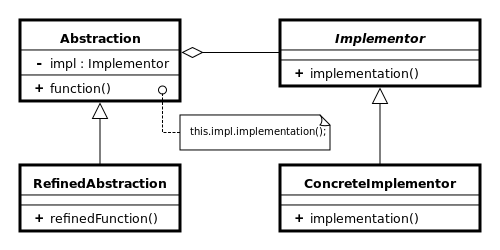
# **Class Diagram**



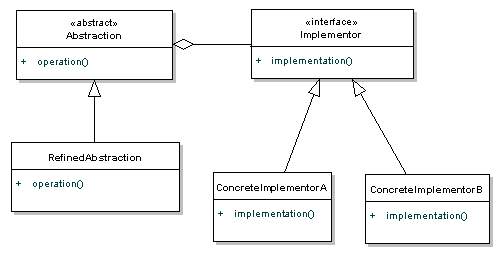
Bridge Design Pattern

GOF: Decouple abstractions from its implementations so that the two can vary independently.

UML Class Diagram



As per DZONE



/\*\* "Implementor" \*/

**interface** IDrawingAPI {

**public** **void** drawCircle();

}

/\*\* "Abstraction" \*/

**public** **abstract** **class** AbstractShape {

**protected** IDrawingAPI drawingAPI;

**protected** AbstractShape(IDrawingAPI drawingAPI) {

**this**.drawingAPI = drawingAPI;

}

**public** **abstract** **void** draw();

}

/\*\* "ConcreteImplementor" 1 \*/

**class** ConcreteDrawingImpl **implements** IDrawingAPI {

**public** **void** drawCircle() {

System.***out***.printf("Drawing circle ... ");

}

}

/\*\* "Refined Abstraction" \*/

**class** CircleShape **extends** AbstractShape {

**public** CircleShape(IDrawingAPI drawingAPI)

{

**super**(drawingAPI);

}

// low-level i.e. Implementation specific

**public** **void** draw() {

drawingAPI.drawCircle();

}

}

**Test Java Program**

/\*\* "Client" \*/

**class** TestBridgePattern {

**public** **static** **void** main(String[] args) {

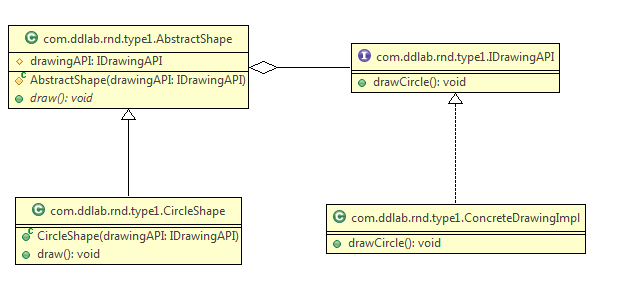
AbstractShape shape = **new** CircleShape(**new** ConcreteDrawingImpl());

shape.draw();

}

}

The class diagram is given below.



**public** **interface** ITV {

**public** **void** on();

**public** **void** off();

**public** **void** switchChannel(**int** channel);

}

**Another example on TV and Remote Control**

**public** **abstract** **class** AbstractRemoteControl {

**private** ITV tv;

**public** AbstractRemoteControl(ITV tv) {

**this**.tv = tv;

}

**public** **void** turnOn(){

tv.on();

}

**public** **void** turnOff(){

tv.off();

}

**public** **void** setChannel(**int** channel){

tv.switchChannel(channel);

}

}

**public** **class** LGTV **implements** ITV {

@Override

**public** **void** on() {

System.***out***.println("Samsung is turned on.");

}

@Override

**public** **void** off() {

System.***out***.println("Samsung is turned off.");

}

@Override

**public** **void** switchChannel(**int** channel) {

System.***out***.println("Samsung: channel - " + channel);

}

}

**public** **class** LGRemoteControl **extends** AbstractRemoteControl {

**public** LGRemoteControl(ITV tv) {

**super**(tv);

}

**public** **void** setChannel(**int** channel){

setChannel(channel);

System.***out***.println("LG uses to set channel.");

}

}

**Test program**

**public** **class** Test {

**public** **static** **void** main(String[] args){

ITV tv = **new** SamsungTV();

LogitechRemoteControl lrc = **new** LogitechRemoteControl(tv);

lrc.setChannel(100);

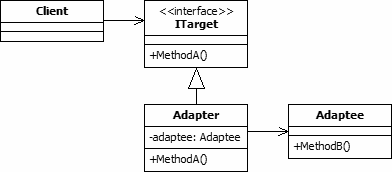
}

}

Adapter Design Pattern

GOF : **Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.**

# **Design Diagram**



**ITarget.java**

public interface ITarget {  
 void MethodA();  
}

Java Code as per the above diagram

**Client.java**

public class Client {  
 private ITarget \_target;  
  
 public Client(ITarget target) {  
 \_target = target;  
 }  
  
 public void MakeRequest() {  
 \_target.MethodA();  
 }  
}

**Adapter.java**

public class Adapter implements ITarget {  
 Adaptee \_adaptee = new Adaptee();  
  
 public void MethodA() {  
 \_adaptee.MethodB();  
 }  
}

**Adaptee.java**

public class Adaptee {  
 public void MethodB() {  
 System.*out*.println("MethodB called");  
 }  
}

**Test1.java**

public class Test1 {  
 public static void main(String[] args) {  
// ITarget target = new LegacyTarget();  
  
 ITarget target = new Adapter();  
  
 Client client = new Client(target); //Fixed  
  
 client.MakeRequest();//Fixed  
 }  
}

**LegacyTarget.java**

public class LegacyTarget implements ITarget {  
  
 @Override  
 public void MethodA() {  
 System.*out*.println("Legacy Method invocation ...");  
 }  
}

An example is given below. An existing system formats the string with new line character. A new system comes which can format the test with csv format.

Java code is given below.

**TextFormatter.java**

//Target  
public interface TextFormatter {  
   
 String formatText(String text);  
}

**Client1.java**

public class Client {  
  
 private TextFormatter formatter;  
  
 public Client(TextFormatter formatter) {  
 this.formatter = formatter;  
 }  
  
 public String format(String text) {  
 return formatter.formatText(text);  
 }  
}

**Test1.java**

public class Test1 {  
 public static void main(String[] args) {  
 String testString = " Formatting line 1. Formatting line 2. Formatting line 3.";  
 TextFormatter textFormatter = new NewLineTextFormatterImpl();  
 // This is as per legacy system  
 String resultString = textFormatter.formatText(testString);  
 System.*out*.println(resultString);  
 }  
}

**CSVFormatterImpl.java**

public class CSVFormatterImpl implements CSVFormatter {  
  
 @Override  
 public String formatCSV(String text) {  
 String formattedText=text.replace(".",",");  
 return formattedText;  
 }  
}

**CSVFormatter.java**

//Adaptee  
public interface CSVFormatter {  
   
 String formatCSV(String text);  
}

**CSVAdapterFormatter.java**

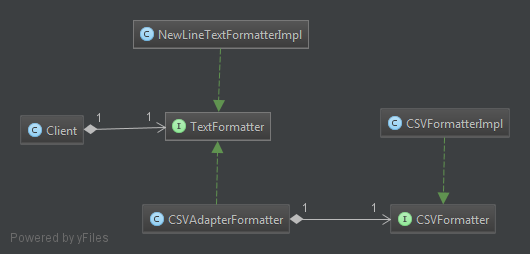
//Adapter  
public class CSVAdapterFormatter implements TextFormatter {  
   
 private CSVFormatter csvFormatter;  
   
 public CSVAdapterFormatter(CSVFormatter csvFormatter) {  
 this.csvFormatter = csvFormatter;  
 }  
  
 @Override  
 public String formatText(String text) {  
 String formattedText=csvFormatter.formatCSV(text);  
 return formattedText;  
 }  
}

The test program is given below.

**Test3.java**

public class Test3 {  
 public static void main(String[] args) {  
 CSVFormatter csvFormatter = new CSVFormatterImpl();  
 TextFormatter textFormatter = new CSVAdapterFormatter(csvFormatter);  
 String testString = " Formatting line 1. Formatting line 2. Formatting line 3.";  
  
 Client client = new Client(textFormatter);  
 String result = client.format(testString);  
 System.*out*.println("Final Result \n"+result);  
 }  
}

The class diagram is given below.



Another Example : Nokia Phone gets changed by Nokia cable and now we have apple cable, we need to use Adapter.

**NonUSBPlug.java**

//Adaptee Interface  
public interface NonUSBPlug {  
 public void connectDevice();  
}

**NokiaPlug.java**

public class NokiaPlug implements Plug {  
  
 @Override  
 public void connect() {  
 System.*out*.println("Nokia plug connected for charging ...");  
 }  
}

NonUSBPlug.java

//Adaptee Imlementation  
public class ApplePlug implements NonUSBPlug {  
  
 @Override  
 public void connectDevice() {  
 System.*out*.println("Device connected for charging ...");  
 }  
}

**MobileClient.java**

//It can be Cell Phone  
public class MobileClient {  
  
 private Plug plug;  
  
 public MobileClient(Plug plug) {  
 this.plug = plug;  
 }  
  
 public void charge() {  
 plug.connect();  
 }  
}

**Plug.java**

public interface Plug {  
 public void connect();  
}

**PlugAdapter.java**

//Adapter  
public class PlugAdapter implements Plug {  
  
 private NonUSBPlug plug;  
  
 public PlugAdapter( NonUSBPlug plug ) {  
 this.plug = plug;  
 }  
  
 @Override  
 public void connect() {  
 plug.connectDevice();  
 }  
}

The test program is given below.

public class Test1 {  
 public static void main(String[] args) {  
  
// NonUSBPlug nonUsbPlug = new ApplePlug();  
// Plug plug = new PlugAdapter(nonUsbPlug);  
  
 Plug plug = new NokiaPlug();  
 MobileClient client = new MobileClient(plug);  
 client.charge();  
 }  
}

Adapter Pattern

<https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/>

We use adapters in our daily lives. The moment you plugin your mobile handset or your laptop to a socket for charging, an adapter is at work. What the adapter does is to make the socket that produces 120 V (or 220 V for European standard) and the mobile device that requires 4 V work together. Similarly, by using the adapter pattern in the programming world, you can make incompatible interfaces work together.

## Participants of the Adapter Pattern

To understand how the adapter pattern works, consider an existing text formatting application comprising of a TextFormattable interface and a NewLineFormatter implementation class. A client provides a string to format with a call to the formatText(String text) method declared in the interface. The implementation class returns a new string by replacing the periods with new line characters.

As an enhancement, the application now needs to support CSV formatting so that the text can be read and edited in a spreadsheet software. The new feature has been implemented and delivered by an external vendor as a CsvFormattable interface along with a CsvFormatter implementation class.

To integrate the new feature in the existing application, several issues require addressing. Primarily, the interface against which the new requirements are implemented has a different structure with different methods, and this is not what the clients of the application expect.

One approach to address such incompatibilities is to ask the vendor to supply a new version with a compatible interface. Another is to update the structure of the existing interface to accommodate the new requirement. Both the approaches will not only result in major rework but also carry the risk of violating the[SOLID](https://springframework.guru/solid-principles-object-oriented-programming/) programming principles, specifically [Single Responsibility Principle](https://springframework.guru/single-responsibility-principle/) and [Interface Segregation Principle](https://springframework.guru/interface-segregation-principle/). In such situations, the adapter pattern comes to the rescue.

Using the adapter pattern, you can create a class, say CsvAdapterImpl that will act as an adapter to make both the incompatible interfaces (TextFormattable and CsvFormattable) work together.

There are two variants of adapters: Object adapter that relies on composition and class adapter that relies on inheritance. As Java does not support multiple inheritance, you have to use object adapter when there are multiple classes that the adapter needs to address. Also, the approach to [favors composition over inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance) should be the driving factor for using object adapters in Java.

We can summarize the participants of the adapter pattern in the context of the text formatting example, as:

* Target (TextFormattable): The existing interface that clients communicate with.
* Adaptee (CsvFormattable): The new incompatible interface that needs adapting.
* Adapter (CsvAdapterImpl): A class that adapts the Adaptee to the Target.
* Client: Communicates with the Target.

## Applying the Adapter Pattern

To apply the adapter pattern to the text formatting example, let’s look at the existing Target interface and its implementation class.

**TextFormattable.java**

public interface TextFormattable {

String formatText(String text);

}

**NewLineFormatter.java**

public class NewLineFormatter implements TextFormattable{

    @Override

    public String formatText(String text)

    {

        String formattedText=text.replace(".","\n");

        return formattedText;

    }

}

In the example above, the TextFormattable interface declares a single formatText() method that the NewLineFormatter class overrides to return a string formatted with new line characters.

Next, let’s look at the Adaptee interface and its implementation class.

**CsvFormattable.java**

package guru.springframework.gof.adapter.adaptee;

public interface CsvFormattable {

String formatCsvText(String text);

}

**CsvFormatter.java**

public class CsvFormatter implements CsvFormattable{

@Override

public String formatCsvText(String text){

String formattedText=text.replace(".",",");

return formattedText;

}

}

The CsvFormattable interface declares a formatCsvText() method that the CsvFormatterclass overrides to return a string formatted as comma separated values. Coming to the main part, we will now write the Adapter class.

**CsvAdapterImpl.java**

package guru.springframework.gof.adapter.csvadapter;

import guru.springframework.gof.adapter.adaptee.CsvFormattable;

import guru.springframework.gof.adapter.source.TextFormattable;

public class CsvAdapterImpl implements TextFormattable {

CsvFormattable csvFormatter;

public CsvAdapterImpl(CsvFormattable csvFormatter){

this.csvFormatter=csvFormatter;

}

@Override

public String formatText(String text)

{

String formattedText=csvFormatter.formatCsvText(text);

return formattedText;

}

}

In the CsvAdapterImpl class above, we implemented the TextFormattable interface, which is the Target. We then declared the Adaptee type (CsvFormattable) as a field and initialized it in the constructor. In the overridden formatText() method, we made a call to the formatCsvText()method, and returned back the CSV formatted string to the caller. Let’s now write a unit test for the example.

**NewLineFormatterTest.java**

import guru.springframework.gof.adapter.adaptee.CsvFormattable;

import guru.springframework.gof.adapter.adaptee.CsvFormatter;

import guru.springframework.gof.adapter.csvadapter.CsvAdapterImpl;

import org.junit.Test;

import static org.junit.Assert.\*;

public class NewLineFormatterTest {

@Test

public void testFormatText() throws Exception {

String testString=" Formatting line 1. Formatting line 2. Formatting line 3.";

TextFormattable newLineFormatter=new NewLineFormatter();

String resultString = newLineFormatter.formatText(testString);

System.out.println(resultString);

CsvFormattable csvFormatter=new CsvFormatter();

TextFormattable csvAdapter=new CsvAdapterImpl(csvFormatter);

String resultCsvString=csvAdapter.formatText(testString);

System.out.println(resultCsvString);

}

}

In Line 17 of the test code above, we called the formatText() method of TextFormattable to format text without using the adapter. It is in Line 20 – Line 22 where we used the adapter. We created a CsvAdapterImpl object passing a CsvFormatter object to its constructor. We then called the formatText() method, which at runtime got forwarded to a call to formatCsvTex() on CsvFormatter.

# Conclusion

Some programmers argue that the Adapter pattern is a workaround for a system, which was not well designed keeping into considerations new possibilities. While this is true to some extent, we cannot expect an enterprise application, which will often have a pluggable architecture, to be designed considering all components that might get added in future.  
In enterprise application development, it is likely that you might need to hook in other libraries, APIs, and “off the shelf” components, and if they are not aligned with the existing system, put the adapter pattern to use. After all, being a core GOF pattern, it is a tested and proven solution used over a long period of time.

Decorator Design Pattern in Java

# **Principle**

**GOF : Attach additional responsibility to an object dynamically. Decorator provides a flexible alternative to subclassing for extending functionality.**

As per Wikipedia , Subclassing adds behavior at [compile time](https://en.wikipedia.org/wiki/Compile_time), and the change affects all instances of the original class; decorating can provide new behavior at [run-time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) for selective objects.

# **Class Diagram**



Java Code given below.

**public** **class** ConcreteComponent **implements** Component {

@Override

**public** String operation() {

**return** "performing operation ….";

}

}

**public** **abstract** **class** Decorator **implements** Component {

**protected** Component component;

**public** Decorator(Component component) {

**this**.component = component;

}

@Override

**public** String operation() {

**return** component.operation();

}

}

**public** **interface** Component {

**public** String operation();

}

**public** **class** ConcreteDecorator **extends** Decorator {

**public** ConcreteDecorator(Component component) {

**super**(component);

}

@Override

**public** String operation() {

**return** **super**.operation()+" "+"something extra";

}

}

**Test Java Program**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Component component = **new** ConcreteDecorator( **new** ConcreteComponent());

System.***out***.println(component.operation());

}

}

**Realtime example : A person makes an order from flipkart for both birthday and marriage gift.**

**public** **class** MobileOrder **implements** Order {

@Override

**public** String getDescription() {

**return** "Mobile";

}

}

**public** **interface** Order {

**public** String getDescription();

}

**public** **abstract** **class** OrderDecorator **implements** Order {

**protected** Order orderTobeDecorated;

**public** OrderDecorator( Order orderTobeDecorated) {

**this**.orderTobeDecorated = orderTobeDecorated;

}

@Override

**public** String getDescription() {

**return** orderTobeDecorated.getDescription();

}

}

**public** **class** BirthDayOrderDecorator **extends** OrderDecorator {

**public** BirthDayOrderDecorator(Order orderTobeDecorated) {

**super**(orderTobeDecorated);

}

@Override

**public** String getDescription() {

**return** **super**.getDescription() + ", including birthday gift wrap";

}

}

**public** **class** MarriageAniversaryOrderDecorator **extends** OrderDecorator {

**public** MarriageAniversaryOrderDecorator(Order orderTobeDecorated) {

**super**(orderTobeDecorated);

}

@Override

**public** String getDescription() {

**return** **super**.getDescription()+" , including Happy Marriage Card";

}

}

Test program

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// Order order = new BirthDayOrderDecorator( new MobileOrder());

//Same day birth day and marriage day

Order order = **new** BirthDayOrderDecorator(**new** MarriageAniversaryOrderDecorator( **new** MobileOrder()));

System.***out***.println(order.getDescription());

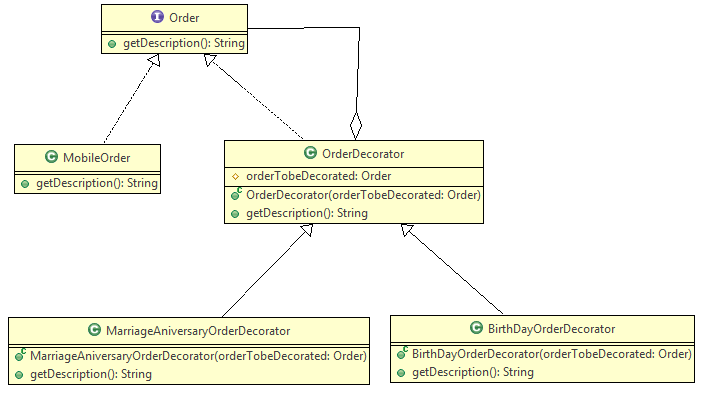
}

}

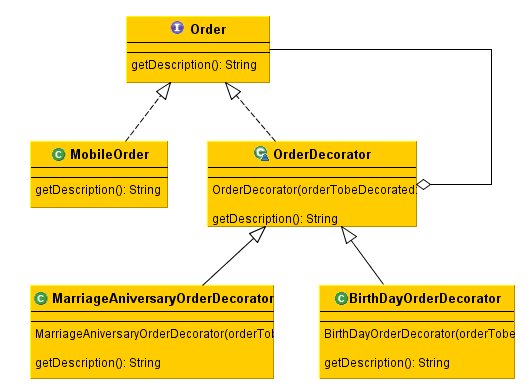
**Output**

Mobile , including Happy Marriage Card, including birthday gift wrap

Class diagram from Eclipse is given below.



Created by YED



**Realtime example : A person opens an account, based upon his popularity he may get some extra benefit like overdraft and multi currency facility.**

**public** **class** RetailAccount **implements** Account {

@Override

**public** String getDescription() {

**return** "Retail account";

}

}

**public** **interface** Account {

**public** String getDescription();

}

**public** **abstract** **class** AccountDecorator **implements** Account {

**protected** Account accountToBeDecorated;

**public** AccountDecorator(Account accountToBeDecorated) {

**this**.accountToBeDecorated = accountToBeDecorated;

}

**public** String getDescription() {

**return** accountToBeDecorated.getDescription(); // Delegation

}

}

**public** **class** MultiCurrencyAccountDecorator **extends** AccountDecorator {

**public** MultiCurrencyAccountDecorator(Account accountToBeDecorated) {

**super**(accountToBeDecorated);

}

@Override

**public** String getDescription() {

**return** **super**.getDescription() + ", including multi currency support";

}

}

**public** **class** OverDraftFacilityAccountDecorator **extends** AccountDecorator {

**public** OverDraftFacilityAccountDecorator(Account accountToBeDecorated) {

**super**(accountToBeDecorated);

}

@Override

**public** String getDescription() {

**return** **super**.getDescription() + ", including overdraft facility";

}

}

**Test Java Program**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Account account = **new** MultiCurrencyAccountDecorator(

**new** OverDraftFacilityAccountDecorator(**new** RetailAccount()));

//Output -> Retail account, including overdraft facility, including multi currency support

// Account account = new MultiCurrencyAccountDecorator(new RetailAccount());

//Output -> Retail account, including multi currency support

// Account account = new OverDraftFacilityAccountDecorator(new RetailAccount());

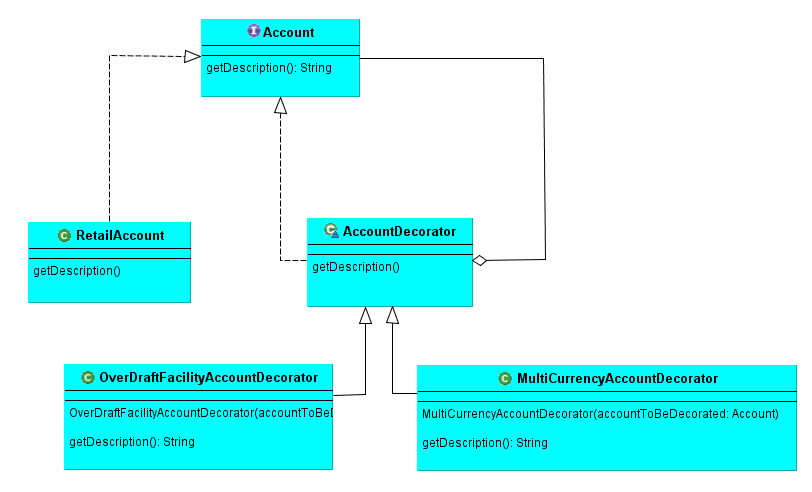
//Output -> Retail account, including overdraft facility

System.***out***.println(account.getDescription());

}

}

UML Class diagram is given below.



AOP using Decorator Pattern

# Introduction

You can also achieve AOP in core java using Decorator Design Pattern. The example is given below.

# AbstractAccount.java

**public abstract class** AbstractAccount {  
  
 **public abstract void** withdraw( String actNo , **int** amount);  
}

# RetailAccount.java

**public class** RetailAccount **extends** AbstractAccount {  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
  
 System.***out***.println(**"Amount "** + amount + **" is debited from the account no "** + actNo);  
 }  
}

# BeforeDecorator.java

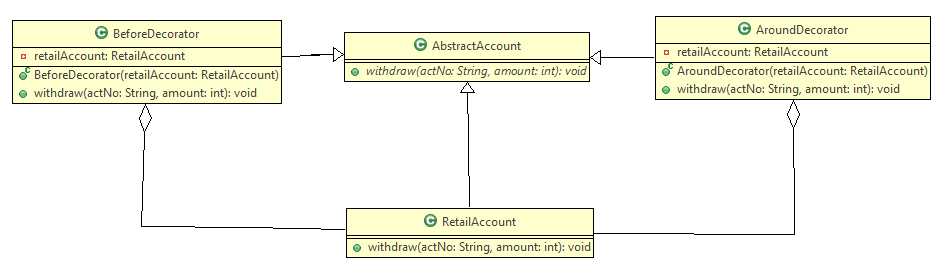
**public class** BeforeDecorator **extends** AbstractAccount {  
  
 **private** RetailAccount **retailAccount**;  
  
 **public** BeforeDecorator(RetailAccount retailAccount) {  
 **this**.**retailAccount** = retailAccount;  
 }  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
 System.***out***.println(**"Doing validation before debiting from acount..."**);  
 **retailAccount**.withdraw(actNo,amount);  
 }  
}

# AroundDecorator.java

**package** com.ddlab.rnd.aop1;  
  
**public class** AroundDecorator **extends** AbstractAccount {  
  
 **private** RetailAccount **retailAccount**;  
  
 **public** AroundDecorator(RetailAccount retailAccount) {  
 **this**.**retailAccount** = retailAccount;  
 }  
  
 @Override  
 **public void** withdraw(String actNo, **int** amount) {  
  
 System.***out***.println(**"Doing before withdrawing money ........."**);  
 **retailAccount**.withdraw(actNo, amount);  
 System.***out***.println(**"Doing after withdrawing money ........."**);  
  
 }  
}

# TestAOPDecorator.java

**package** com.ddlab.rnd.aop1;  
  
**public class** TestAOPDecorator {  
  
 **public static void** main(String[] args) {  
  
 RetailAccount account = **new** RetailAccount();  
*// BeforeDecorator beforeDecorator = new BeforeDecorator(account);* AroundDecorator aroundDecorator = **new** AroundDecorator(account);  
 aroundDecorator.withdraw(**"1122334455"**,500);  
 }  
}



Decorator Design Pattern

According GOF

**Intent**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

### Motivation

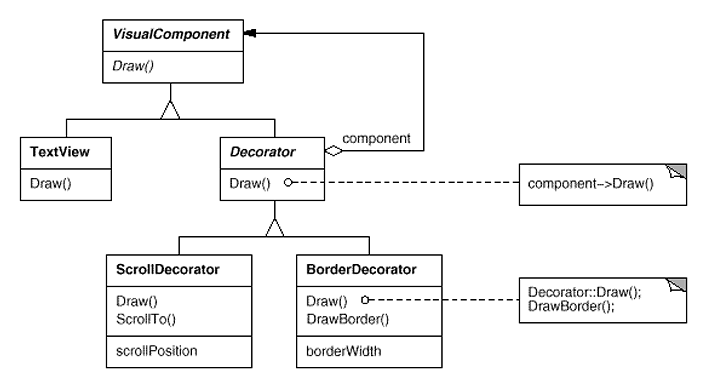
Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

### UML

The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.



VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface. Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.

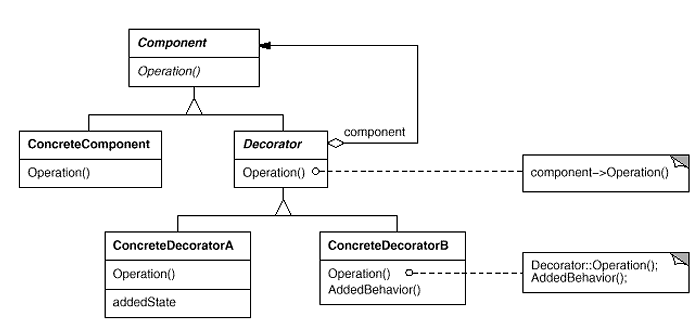
Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface if they know there happens to be a ScrollDecorator object in the interface. The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.

### Applicability

Use Decorator

* to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
* for responsibilities that can be withdrawn.
* when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

### Actual UML Diagram



**Participants**

* **Component** (VisualComponent)
  + defines the interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent** (TextView)
  + defines an object to which additional responsibilities can be attached.
* **Decorator**
  + maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  + adds responsibilities to the component.

**Collaborations**

* Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

**Consequences**

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance.* The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.

Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.

1. *Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.
2. *A decorator and its component aren't identical.* A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.
3. *Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

### Related Patterns

**Adapter** : A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

**Composite** : A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

**Strategy** : A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

# Source Code

## Type-1

**package** com.design.pattern.decorator.type1;

**public** **interface** **Shape** {

**void** draw();

}

**package** com.design.pattern.decorator.type1;

**public** **class** **Circle** **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Shape: Circle");

}

}

**package** com.design.pattern.decorator.type1;

**public** **class** **Rectangle** **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Shape: Rectangle");

}

}

**package** com.design.pattern.decorator.type1;

**public** **abstract** **class** **ShapeDecorator** **implements** Shape {

**protected** Shape decoratedShape;

**public** ShapeDecorator(Shape decoratedShape){

**this**.decoratedShape = decoratedShape;

}

**public** **void** draw(){

decoratedShape.draw();

}

}

**package** com.design.pattern.decorator.type1;

**public** **class** **RedShapeDecorator** **extends** ShapeDecorator {

**public** RedShapeDecorator(Shape decoratedShape) {

**super**(decoratedShape);

}

@Override

**public** **void** draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

**private** **void** setRedBorder(Shape decoratedShape){

System.***out***.println("Border Color: Red");

}

}

**package** com.design.pattern.decorator.type1;

**public** **class** **DecoratorPatternDemo** {

**public** **static** **void** main(String[] args) {

Shape circle = **new** Circle();

//Normal Circle

System.***out***.println("Normal Circle");

circle.draw();

System.***out***.println("Circle with border");

Shape redCircle = **new** RedShapeDecorator(circle);

redCircle.draw();

Shape redRectangle = **new** RedShapeDecorator(**new** Rectangle());

System.***out***.println("\nRectangle of red border");

redRectangle.draw();

}

}

## Type-2

**package** com.design.pattern.decorator.type2;

**public** **interface** **IEmail**

{

**public** String getContents();

}

**package** com.design.pattern.decorator.type2;

//concrete component

**public** **class** **Email** **implements** IEmail

{

**private** String content;

**public** Email(String content)

{

**this**.content = content;

}

@Override

**public** String getContents()

{

//general email stuff

**return** content;

}

}

**package** com.design.pattern.decorator.type2;

**public** **abstract** **class** **EmailDecorator** **implements** IEmail

{

//wrapped component

IEmail originalEmail;

@Override

**public** String getContents() {

**return** originalEmail.getContents();

}

}

**package** com.design.pattern.decorator.type2;

//concrete decorator

**public** **class** **ExternalEmailDecorator** **extends** EmailDecorator

{

**private** String content;

**public** ExternalEmailDecorator(IEmail basicEmail)

{

originalEmail = basicEmail;

}

@Override

**public** String getContents()

{

// secure original

content = addDisclaimer(originalEmail.getContents());

**return** content;

}

**private** String addDisclaimer(String message)

{

//append company disclaimer to message

**return** message + "\n Company Disclaimer";

}

}

**package** com.design.pattern.decorator.type2;

**public** **class** **EmailSender**

{

**public** **void** sendEmail(IEmail email)

{

//read the email to-address, to see if it's going outside of the company

//if so decorate it

IEmail external = **new** ExternalEmailDecorator(email);

System.***out***.println(external.getContents());

//send

}

**public** **static** **void** main(String[] args) {

EmailSender mail = **new** EmailSender();

IEmail email1 = **new** Email("ssadasds");

mail.sendEmail(email1);

}

}

## Type – 3

**package** com.design.pattern.decorator.type3;

**public** **interface** **Car** {

**public** **void** assemble();

}

**package** com.design.pattern.decorator.type3;

**public** **class** **BasicCar** **implements** Car {

@Override

**public** **void** assemble() {

System.***out***.print("Basic Car.");

}

}

**package** com.design.pattern.decorator.type3;

**public** **abstract** **class** **CarDecorator** **implements** Car {

**protected** Car car;

**public** CarDecorator(Car c){

**this**.car=c;

}

@Override

**public** **void** assemble() {

**this**.car.assemble();

}

}

**package** com.design.pattern.decorator.type3;

**public** **class** **LuxuryCar** **extends** CarDecorator {

**public** LuxuryCar(Car c) {

**super**(c);

}

@Override

**public** **void** assemble(){

car.assemble();

System.***out***.print(" Adding features of Luxury Car.");

}

}

**package** com.design.pattern.decorator.type3;

**public** **class** **SportsCar** **extends** CarDecorator {

**public** SportsCar(Car c) {

**super**(c);

}

@Override

**public** **void** assemble(){

car.assemble();

System.***out***.print(" Adding features of Sports Car.");

}

}

**package** com.design.pattern.decorator.type3;

**public** **class** **DecoratorPatternTest** {

**public** **static** **void** main(String[] args) {

Car sportsCar = **new** SportsCar(**new** BasicCar());

sportsCar.assemble();

System.***out***.println("\n\*\*\*\*\*");

Car sportsLuxuryCar = **new** SportsCar(**new** LuxuryCar(**new** BasicCar()));

sportsLuxuryCar.assemble();

}

}

# Type – 4 – From Wikipedia

**package** com.design.pattern.decorator.type4;

**public** **interface** **Window** {

**public** **void** draw(); // Draws the Window

**public** String getDescription(); // Returns a description of the Window

}

**package** com.design.pattern.decorator.type4;

**class** **SimpleWindow** **implements** Window {

**public** **void** draw() {

// Draw window

}

**public** String getDescription() {

**return** "simple window";

}

}

**package** com.design.pattern.decorator.type4;

**class** **HorizontalScrollBarDecorator** **extends** WindowDecorator {

**public** HorizontalScrollBarDecorator (Window windowToBeDecorated) {

**super**(windowToBeDecorated);

}

@Override

**public** **void** draw() {

**super**.draw();

drawHorizontalScrollBar();

}

**private** **void** drawHorizontalScrollBar() {

// Draw the horizontal scrollbar

}

@Override

**public** String getDescription() {

**return** **super**.getDescription() + ", including horizontal scrollbars";

}

}

**package** com.design.pattern.decorator.type4;

**class** **VerticalScrollBarDecorator** **extends** WindowDecorator {

**public** VerticalScrollBarDecorator (Window windowToBeDecorated) {

**super**(windowToBeDecorated);

}

@Override

**public** **void** draw() {

**super**.draw();

drawVerticalScrollBar();

}

**private** **void** drawVerticalScrollBar() {

// Draw the vertical scrollbar

}

@Override

**public** String getDescription() {

**return** **super**.getDescription() + ", including vertical scrollbars";

}

}

**package** com.design.pattern.decorator.type4;

**abstract** **class** **WindowDecorator** **implements** Window {

**protected** Window windowToBeDecorated; // the Window being decorated

**public** WindowDecorator (Window windowToBeDecorated) {

**this**.windowToBeDecorated = windowToBeDecorated;

}

**public** **void** draw() {

windowToBeDecorated.draw(); //Delegation

}

**public** String getDescription() {

**return** windowToBeDecorated.getDescription(); //Delegation

}

}

**package** com.design.pattern.decorator.type4;

**public** **class** **DecoratedWindowTest** {

**public** **static** **void** main(String[] args) {

// Create a decorated Window with horizontal and vertical scrollbars

Window decoratedWindow = **new** HorizontalScrollBarDecorator (

**new** VerticalScrollBarDecorator (**new** SimpleWindow()));

// Print the Window's description

System.***out***.println(decoratedWindow.getDescription());

}

}

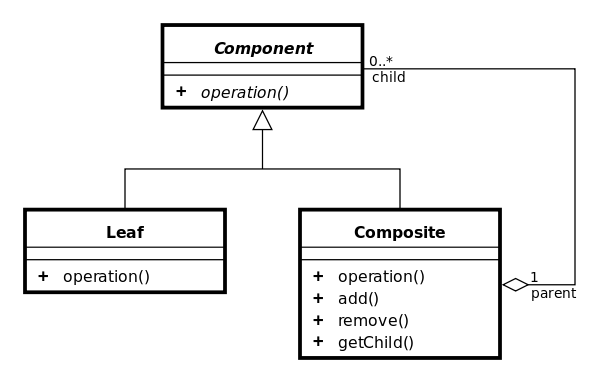
Similarly you can provide the following example

* + 1. Assault rifle with telescope and night vision
    2. Order management with premium order with gift wrap.
    3. Generation of monthly report and yearly report.
    4. Credit Card with micro chip.
    5. Car with bullet proof.

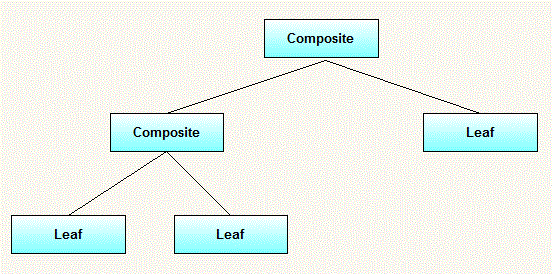
Composite Design Pattern

**GOF : Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly**.

# **Class Diagram**



Tree structure is given below.



Example : Directory and File, both have the ls() method. Java code is given below.

**class** File **implements** AbstractFile {

**private** String name;

**public** File(String name) {

**this**.name = name;

}

**public** **void** ls() {

System.***out***.println(name);

}

}

**import** java.util.ArrayList;

**public** **class** Directory **implements** AbstractFile {

**private** String name;

**private** ArrayList<AbstractFile> files = **new** ArrayList<AbstractFile>();

**public** Directory(String name) {

**this**.name = name;

}

**public** **void** add(AbstractFile f) {

files.add(f);

}

**public** **void** ls() {

System.***out***.println(name);

**for** (AbstractFile file : files) {

file.ls();

}

}

}

**public** **interface** AbstractFile {

**public** **void** ls();

}

The test program is given below.

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Directory dirOne = **new** Directory("dir111");

Directory dirTwo = **new** Directory("dir222");

File a = **new** File("a");

File b = **new** File("b");

File c = **new** File("c");

File d = **new** File("d");

dirOne.add(a);

dirOne.add(dirTwo);

dirOne.add(b);

dirTwo.add(c);

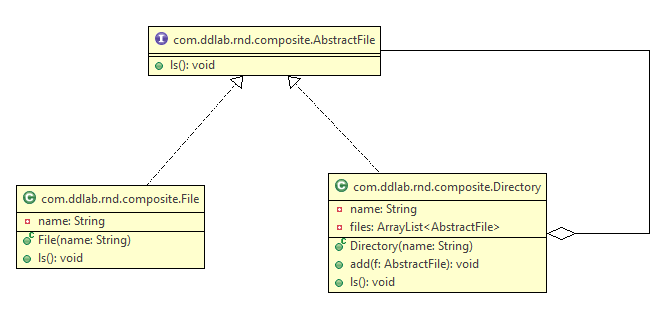
dirTwo.add(d);

dirOne.ls();

}

}

**The UML class diagram is given below.**

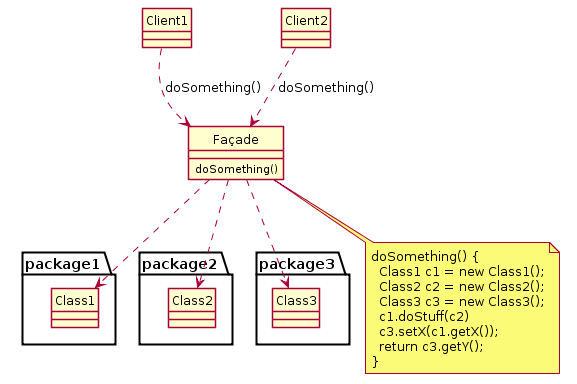


Another example : Manager and Employee.

Façade Design Pattern

GOF : **Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.**

# **Class Diagram**



# **Java Code**

class CPU {

public void freeze() { ... }

public void jump(long position) { ... }

public void execute() { ... }

}

class Memory {

public void load(long position, byte[] data) { ... }

}

class HardDrive {

public byte[] read(long lba, int size) { ... }

}

*/\* Facade \*/*

class ComputerFacade {

private CPU processor;

private Memory ram;

private HardDrive hd;

public ComputerFacade() {

this.processor = new CPU();

this.ram = new Memory();

this.hd = new HardDrive();

}

public void start() {

processor.freeze();

ram.load(BOOT\_ADDRESS, hd.read(BOOT\_SECTOR, SECTOR\_SIZE));

processor.jump(BOOT\_ADDRESS);

processor.execute();

}

}

*/\* Client \*/*

class You {

public static void main(String[] args) {

ComputerFacade computer = new ComputerFacade();

computer.start();

}

}

# **Another Example**

public class Façade {

    public void PerformAction() {

        Class1A c1a = new Class1A();

        Class1B c1b = new Class1B();

        Class2A c2a = new Class2A();

        Class2B c2b = new Class2B();

        int result1a = c1a.Method1A();

        int result1b = c1b.Method1B(result1a);

        int result2a = c2a.Method2A(result1a);

        c2b.Method2B(result1b, result2a);

    }

}

public class Class1A {

    public int Method1A() {

        // Sample code

    }

}

public class Class1B {

    public int Method1B(int param) {

        // Sample code

    }

}

public class Class2A {

    public int Method2A(int param) {

        // Sample code

    }

}

public class Class2B {

    public void Method2B(int param1, int param2) {

        // Sample code

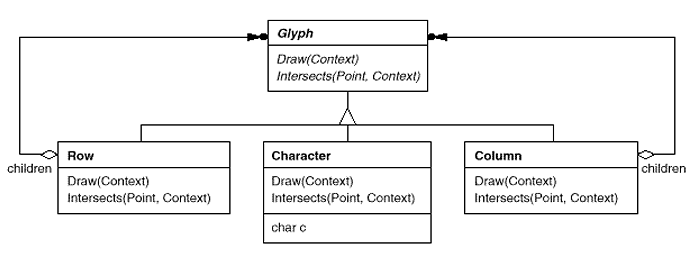
    }

}

Flyweight Design Pattern

GOF : **Use sharing to support large numbers of fine-grained objects efficiently.**

# **Class Diagram**



As suggested by Joshua Bloch, you can achieve Flyweight Design using static factory method. Java language

uses Boolean where the **valueOf()** method properly uses Flyweight Design and returns the similar Boolean object

without creating repeatedly. For the convenience, the code structure is given below.

public final class Boolean implements java.io.Serializable,Comparable<Boolean>

{

public static final Boolean TRUE = new Boolean(true);

public static final Boolean FALSE = new Boolean(false);

// Use of Flyweight Design Pattern

public static Boolean valueOf(boolean b)

{

return (b ? TRUE : FALSE);

}

}

The class **java.util.EnumSet (Item 32),** introduced in release 1.5, has no public constructors, only static factories. They return one of two implementations, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a **RegularEnumSet** instance, which is backed by a single long; if the enum type has sixty-five or more elements, the factories return a **JumboEnumSet** instance, backed by a long array.

# **FlyWeight Pattern Example**

Let us consider we have three kinds of Cards like MASTER,VISA,ELECTRON. We have to choose the card based on the bin of the card without recreating the objects.

public final class Card

{

private String bin = null;

private final static Card MASTER = new Card("1111");

private final static Card VISA = new Card("2222");

private final static Card ELECTRON = new Card("3333");

private final static Card INVALID = new Card("4444");

private Card(String bin)

{

this.bin = bin;

}

public static Card valueOf( String cardNo )

{

Card card = null;

if( cardNo.startsWith("1111")) card = MASTER;

else if( cardNo.startsWith("2222")) card = VISA;

else if( cardNo.startsWith("3333")) card = ELECTRON;

else card = INVALID;

return card;

}

public String getTransactionDetails()

{

return "Transaction Details for "+toString();

}

@Override

public String toString()

{

String cardType = null;

if( bin.startsWith("1111")) cardType = "MASTER";

else if( bin.startsWith("2222")) cardType = "VISA";

else if( bin.startsWith("3333")) cardType = "ELECTRON";

else cardType = "INVALID";

return cardType;

}

}

package com.type2;

public class TestCard

{

public static void main(String[] args)

{

String cardNo = "3333 1111 2222 4444";

Card card = Card.valueOf(cardNo);

System.out.println("Card Type ::: "+card.toString());

System.out.println(card.getTransactionDetails());

}

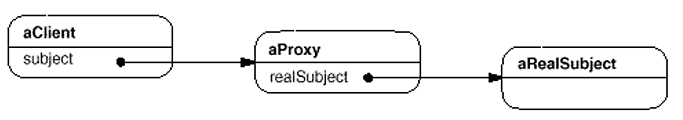
}

Proxy Design Pattern

GOF : **Provide a surrogate or placeholder for another object to control access to it.**

The main benefit behind proxy design pattern is minimize the usage of very large or complex object or impossible object.

# **Object Diagram**



Let us consider a basic example.

interface Image {

public abstract void displayImage();

}

public class RealImage implements Image {

private String filename = null;

/\*\*

\* Constructor

\* @param FILENAME

\*/

public RealImage(final String FILENAME) {

filename = FILENAME;

loadImageFromDisk();

}

/\*\*

\* Loads the image from the disk

\*/

private void loadImageFromDisk() {

System.out.println("Loading " + filename);

}

/\*\*

\* Displays the image

\*/

public void displayImage() {

System.out.println("Displaying " + filename);

}

}

public class ProxyImage implements Image {

private RealImage image = null;

private String filename = null;

/\*\*

\* Constructor

\* @param FILENAME

\*/

public ProxyImage(final String FILENAME) {

filename = FILENAME;

}

/\*\*

\* Displays the image

\*/

public void displayImage() {

if (image == null) {

image = new RealImage(filename);

}

image.displayImage();

}

}

class TestProxyDesign {

/\*\*

\* Test method

\*/

public static void main(String[] args) {

final Image IMAGE1 = new ProxyImage("HiRes\_10MB\_Photo1");

final Image IMAGE2 = new ProxyImage("HiRes\_10MB\_Photo2");

IMAGE1.displayImage(); // loading necessary

IMAGE1.displayImage(); // loading unnecessary

IMAGE2.displayImage(); // loading necessary

IMAGE2.displayImage(); // loading unnecessary

IMAGE1.displayImage(); // loading unnecessary

}

}

In the above case, there is no need to create the Image object once again.

Let us corelate with the real world objects and entities. As we know that vendors like Dell, Lenovo, HP can not be reached directly. If you want to get technical solution from them, you have to contact their Call Centres or Customer Care. In this case Customer Care or Call Centres are the proxies on behalf of the respective vendors.

Let us consider another good example. As you know that it is very difficult to meet the President of a Nation directly. We have to go through the channel or we have to meet the PA to President. In this PA to President is the Proxy on behalf of President of the Nation.

Let us consider the following piece of code.

public interface SolutionProvider {

public String provideSolution();

}

Public final class President implements SolutionProvider {

@Override

public String provideSolution() {

System.out.println("Solution Provided by President of the Nation");

return "Solution Provided";

}

}

public class ProxyPresident implements SolutionProvider {

private String problemType;

private President president = new President();

public ProxyPresident( String problemType ) {

this.problemType = problemType;

}

@Override

public String provideSolution() {

if( problemType.equals("Political") || problemType.equals("Economic") )

return president.provideSolution();

else

{

System.out.println("Solution Provided by Proxy which is a PA to President");

return "Solution Provided";

}

}

}

The test harnes class is given below.

import com.ddlab.design.proxy.type2.ProxyPresident;

import com.ddlab.design.proxy.type2.SolutionProvider;

public class TestProxyPattern {

public static void main(String[] args) {

String problemType = "Love Issue";

SolutionProvider provider = new ProxyPresident(problemType);

String solution = provider.provideSolution();

System.out.println("Solution : "+solution);

problemType = "Political";

provider = new ProxyPresident(problemType);

solution = provider.provideSolution();

System.out.println("Solution : "+solution);

}

}

From the above design of the class, it is evident that you can not directly meet the president as the class is

package private, ofcourse you can not extend it. The most import point here is that if you have any issue related to Love or relationship, the solution will be provided by Proxy which is the PA to President other political and economic issues will be resolved only by President.

There is another good example of Proxy design suggested by Joshua Bloch is SerializationProxy.

package org.effectivejava.examples.chapter11.item78;

import java.io.InvalidObjectException;

import java.io.ObjectInputStream;

import java.io.Serializable;

import java.util.Date;

public final class Period implements Serializable {

private final Date start;

private final Date end;

public Period(Date start, Date end) {

this.start = new Date(start.getTime());

this.end = new Date(end.getTime());

if (this.start.compareTo(this.end) > 0)

throw new IllegalArgumentException(start + " after " + end);

}

public Date start() {

return new Date(start.getTime());

}

public Date end() {

return new Date(end.getTime());

}

public String toString() {

return start + " - " + end;

}

// Serialization proxy for Period class - page 312

private static class SerializationProxy implements Serializable {

private final Date start;

private final Date end;

SerializationProxy(Period p) {

this.start = p.start;

this.end = p.end;

}

//use Any Number

private static final long serialVersionUID = 234098243823485285L;

// readResolve method for Period.SerializationProxy - Page 313

private Object readResolve() {

return new Period(start, end); // Uses public constructor

}

}

// writeReplace method for the serialization proxy pattern - page 312

private Object writeReplace() {

return new SerializationProxy(this);

}

// readObject method for the serialization proxy pattern - Page 313

private void readObject(ObjectInputStream stream)

throws InvalidObjectException {

throw new InvalidObjectException("Proxy required");

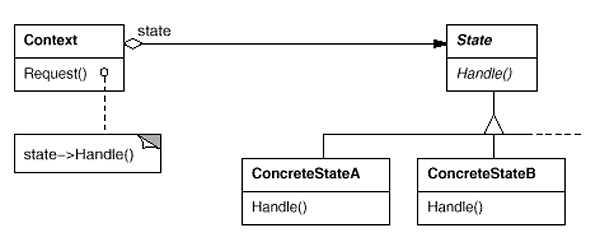
}

}

State Design Pattern

GOF : **Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.**

**Structure Diagram**



Java Code : Example is mobile state silent, vibration

**public** **interface** MobileState {

**void** display(Context context);

}

**public** **class** Context {

**private** MobileState state;

**public** MobileState getState() {

**return** state;

}

**public** **void** setState(MobileState state) {

**this**.state = state;

}

**public** **void** request() {

state.display(**this**);

}

}

Class Diagram

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Context context = **new** Context();

context.setState( **new** SilentState());

context.request();

context.setState( **new** VibrationState());

context.request();

}

}

**public** **class** SilentState **implements** MobileState {

@Override

**public** **void** display(Context context) {

System.***out***.println("Silent...");

}

}

**public** **class** VibrationState **implements** MobileState {

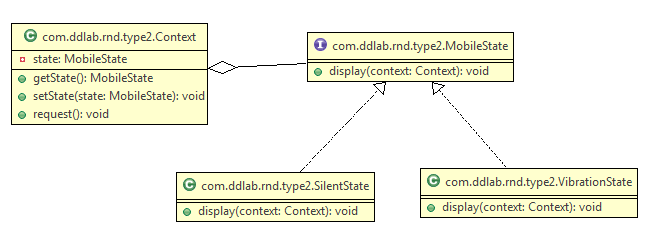
@Override

**public** **void** display(Context context) {

System.***out***.println("vibration...");

}

}



State Design Pattern Example

**public** **class** Driver {

**public** **static** **void** main(String[] args) {

CarContext car = **new** CarContext();

AccelerationState normal = **new** NormalAccelerationState();

AccelerationState turbo = **new** TurboAccelerationState();

car.setAccelerationState(normal);

car.accelerate();

System.***out***.println("Current speed: " + car.getSpeed());

car.setAccelerationState(turbo);

car.accelerate();

System.***out***.println("Current speed: " + car.getSpeed());

}

}

//Context

**public** **class** CarContext {

**private** AccelerationState accelerationState;

**private** **int** speed = 0;

**public** **void** setAccelerationState(AccelerationState accelerationState) {

**this**.accelerationState = accelerationState;

}

**public** **void** accelerate() {

speed += accelerationState.accelerate();

}

**public** **int** getSpeed() {

**return** speed;

}

}

//Concrete State

**public** **class** TurboAccelerationState **implements** AccelerationState {

@Override

**public** **int** accelerate() {

**return** 10;

}

}

//Concrete State

**public** **class** NormalAccelerationState **implements** AccelerationState {

@Override

**public** **int** accelerate() {

**return** 5;

}

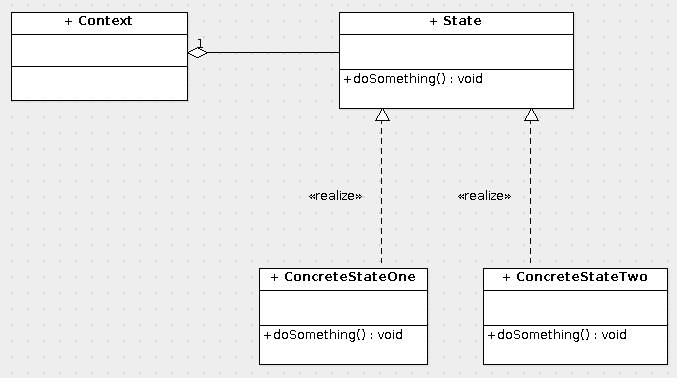
}

//State

**public** **interface** AccelerationState {

**int** accelerate();

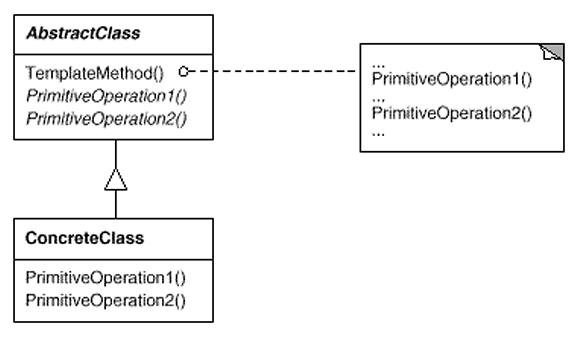
}



Template Design Pattern

GOF: **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.**

**Structure Diagram**



Java Code Example

**public** **class** MonthlyReport **extends** AbstractReport {

@Override

**public** **void** setHeader() {

System.***out***.println("Header set ...");

}

@Override

**public** **void** setBody() {

System.***out***.println("Body set ...");

}

@Override

**public** **void** setFooter() {

System.***out***.println("Footer set ...");

}

}

**public** **abstract** **class** AbstractReport {

**public** **abstract** **void** setHeader();

**public** **abstract** **void** setBody();

**public** **abstract** **void** setFooter();

**public** **void** generateReport() {

setHeader();

setBody();

setFooter();

}

}

Class Diagram

**public** **class** Test {

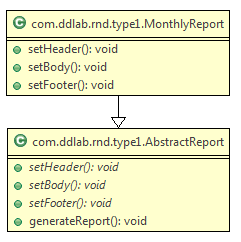
**public** **static** **void** main(String[] args) {

AbstractReport report = **new** MonthlyReport();

report.generateReport();

}

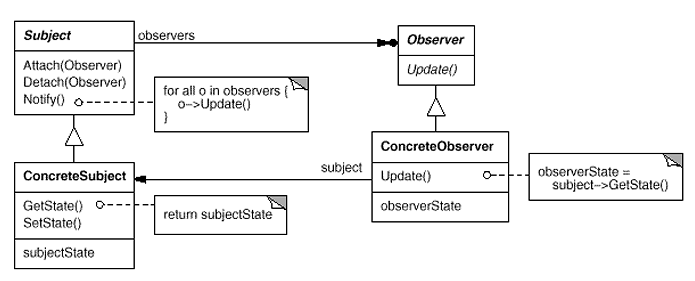
}



Observer Design Pattern

GOF : **Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically**.

**Structure Diagram**



**Basic Java Code**

**Observer.java**

**public** **abstract** **class** Observer {

**protected** Subject subject;

**public** **abstract** **void** update();

}

**Subject.java**

**import** java.util.ArrayList;

**import** java.util.List;

**class** Subject {

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

**private** String state;

**public** **void** attach(Observer o) {

observers.add(o);

}

**public** String getState() {

**return** state;

}

**public** **void** setState(String in) {

state = in;

notifyObservers();

}

**private** **void** notifyObservers() {

**for**( Observer observer : observers )

observer.update();

}

}

**ConcreteObserver1.java**

**public** **class** ConcreteObserver1 **extends** Observer {

**public** ConcreteObserver1(Subject s) {

subject = s;

subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("ConcreteObserver1 got updated state : "+subject.getState());

}

}

**ConcreteObser2.java**

**public** **class** ConcreteObserver2 **extends** Observer {

**public** ConcreteObserver2(Subject s) {

subject = s;

subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("ConcreteObserver2 got updated state : "+subject.getState());

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Subject subject = **new** Subject();

**new** ConcreteObserver1(subject);

**new** ConcreteObserver2(subject);

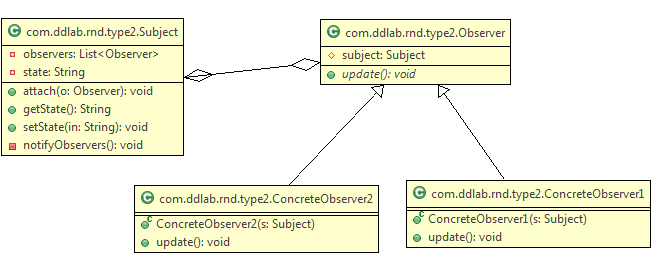
subject.setState("Delhi");

subject.setState("Bangalore");

}

}

**UML Diagram**

****

No let us convert the above example of Aeroplane and AirTrafficControllers.

**AbstractTrafficController.java**

**public** **abstract** **class** AbstractTrafficController {

**protected** Aeroplane plane;

**public** **abstract** **void** update();

}

**AirTrafficController.java**

**public** **class** AirTrafficController **extends** AbstractTrafficController {

**public** AirTrafficController(Aeroplane s) {

plane = s;

plane.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***

.println("AirTrafficController got info that , plane has come to : "

+ plane.getTravelledCity());

}

}

**Aeroplane.java**

**import** java.util.ArrayList;

**import** java.util.List;

**class** Aeroplane {

**private** List<AbstractTrafficController> observers = **new** ArrayList<AbstractTrafficController>();

**private** String travelledCity;

**public** **void** attach(AbstractTrafficController o) {

observers.add(o);

}

**public** String getTravelledCity() {

**return** travelledCity;

}

**public** **void** setTravelledCity(String travelledCity) {

**this**.travelledCity = travelledCity;

notifyObservers();

}

**private** **void** notifyObservers() {

**for** (AbstractTrafficController observer : observers)

observer.update();

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Aeroplane plane = **new** Aeroplane();

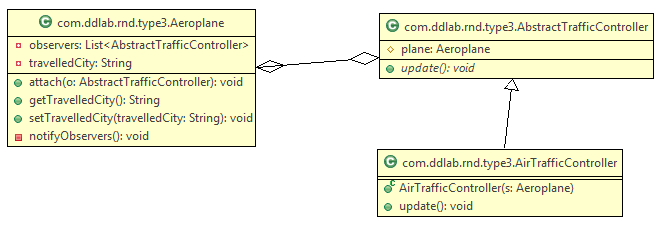
**new** AirTrafficController(plane);

plane.setTravelledCity("Delhi");

}

}

**UML Diagram**

****

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern helps in maintaining consistency between related objects without making classes tightly coupled. Let us consider an example that an aeroplane is monitored by Air Traffic Controller . Air Traffic contoller will monitor the city the aeroplance has travelled or crossed. Let us have a look at the following example.

In this case Aeroplane extends Observable whereas Air Traffic Contoller implements Observer as Air Traffic contoller wants to observer. So Air Traffic Contoller is an observer. This pattern helps to know the recent update of an object state.

import java.util.Observable;

public class Aeroplane extends Observable

{

private String travelledCity;

public String getTravelledCity() {

return travelledCity;

}

public void setTravelledCity(String travelledCity) {

this.travelledCity = travelledCity;

setChanged();

notifyObservers(travelledCity);

}

}

import java.util.Observable;

import java.util.Observer;

public class AirTrafficController implements Observer

{

private String cityName;

@Override

public void update(Observable o, Object arg)

{

System.out.println("The Aeroplane has crossed the city "+arg);

cityName = (String)arg;

}

public String getCityName() {

return cityName;

}

}

public class MonitorAeroplaneTest {

public static void main(String[] args) {

Aeroplane aeroplane = new Aeroplane();

AirTrafficController atc = new AirTrafficController();

aeroplane.addObserver(atc);

aeroplane.setTravelledCity("India");

System.out.println("Aeroplane is travelling towards : "+atc.getCityName());

aeroplane.setTravelledCity("Moscow");

System.out.println("Aeroplane is travelling towards : "+atc.getCityName());

}

}

Similarly you can think of an examples.

1. Service provider tracks the location of the mobile phone.

2. In case banking customer monitors the changes in the account.

Similarly you can achieve the above using ProertyChangeListener. Let us have a look.

import java.beans.PropertyChangeEvent;

import java.beans.PropertyChangeListener;

import java.beans.PropertyChangeSupport;

public class MyAccount implements PropertyChangeListener

{

private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

private int amount = 5000;

public MyAccount() {

pcs.addPropertyChangeListener(this);

}

public int getAmount() {

return amount;

}

public void setAmount(int amount) {

pcs.firePropertyChange("amount", this.amount, amount);

this.amount = amount;

}

public void withDrawAmount() {

System.out.println("Amount is withdrawn ...");

}

@Override

public void propertyChange(PropertyChangeEvent evt) {

System.out.println("Name = " + evt.getPropertyName());

System.out.println("Old Value = " + evt.getOldValue());

System.out.println("New Value = " + evt.getNewValue());

}

}

public class TestAccount {

public static void main(String[] args) {

MyAccount act = new MyAccount();

act.setAmount(111);

act.withDrawAmount();

act.setAmount(11);

act.withDrawAmount();

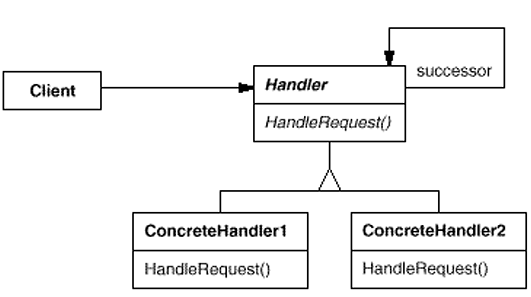
}

**}**

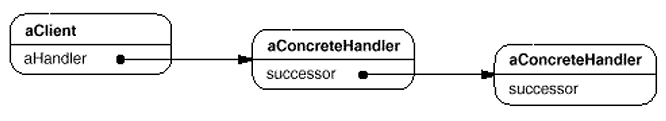
Chain of Responsibility

GOF : **Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.**

# **Structure Diagram**



Object Structure Diagam



**Java Code Example :** A loan has be to approved based upon the amount by various actors like LoanOfficer, BranchManager and Vice President of the Bank.

**LoanData.java**

**public** **class** LoanData {

**private** String purpose;

**private** **int** amount;

**public** LoanData(String purpose, **int** amount) {

**this**.purpose = purpose;

**this**.amount = amount;

}

**public** String getPurpose() {

**return** purpose;

}

**public** **int** getAmount() {

**return** amount;

}

}

**LoanApprover.java**

**public** **abstract** **class** LoanApprover {

**protected** LoanApprover successor;

**public** **void** setSuccessor(LoanApprover successor) {

**this**.successor = successor;

}

**public** **abstract** **void** approveLoan(LoanData data);

}

**LoanOfficer.java**

**public** **class** LoanOfficer **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

**if**( data.getAmount() < 1000 )

System.***out***.println("Loan approved by LoanOfficer ...");

**else**

successor.approveLoan(data);

}

}

**BranchManager.java**

**public** **class** BranchManager **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

**if**( data.getAmount() > 1000 && data.getAmount() < 10000 )

System.***out***.println("Loan approved by BranchManager ...");

**else**

successor.approveLoan(data);

}

}

**VicePresident.java**

**public** **class** VicePresident **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

System.***out***.println("Loan approved by VicePresident ...");

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

LoanData data = **new** LoanData("Cultivation", 50000);

LoanApprover loanOfficer = **new** LoanOfficer();

LoanApprover manager = **new** BranchManager();

LoanApprover vp = **new** VicePresident();

//Form the chain

loanOfficer.setSuccessor(manager);

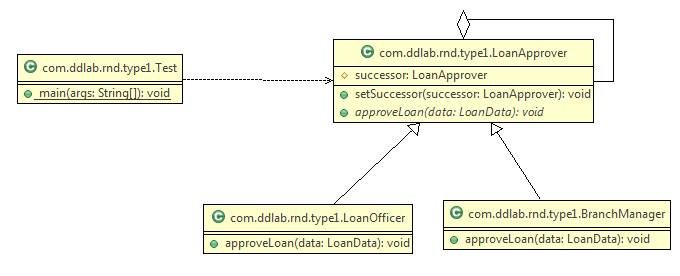
manager.setSuccessor(vp);

loanOfficer.approveLoan(data);

}

}

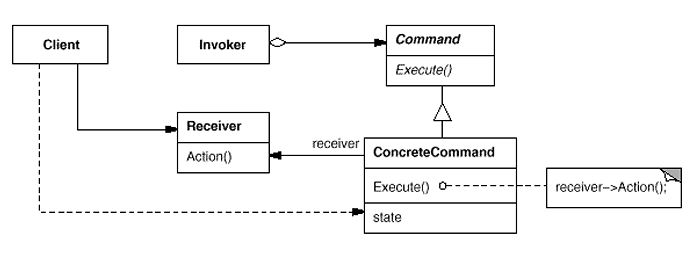
**UML Class Diagram** is given below.



Command Pattern

GOF: **Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations**.

# **Structure Diagram**

  
Java Code Example : To switch off and switch on light.

**Command.java**

**LightOnCommand.java**

//Concrete Command

**public** **class** LightOnConcreteCommand **implements** Command {

// reference to the light

LightReceiver light;

**public** LightOnConcreteCommand(LightReceiver light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOn();

}

}

//Command

**public** **interface** Command{

**public** **void** execute();

}

**LightReceiver.java**

//Receiver

**public** **class** LightReceiver {

**private** **boolean** on;

**public** **void** switchOn() {

on = **true**;

}

**public** **void** switchOff() {

on = **false**;

}

}

**LightOffConcreteCommand.java**

//Concrete Command

**public** **class** LightOffConcreteCommand **implements** Command {

// reference to the light

LightReceiver light;

**public** LightOffConcreteCommand(LightReceiver light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOff();

}

}

**RemoteController.java**

//Invoker

**public** **class** RemoteControlInvoker {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** pressButton() {

command.execute();

}

}

**Client.java**

//Client

**public** **class** Client {

**public** **static** **void** main(String[] args) {

RemoteControlInvoker control = **new** RemoteControlInvoker();

LightReceiver light = **new** LightReceiver();

Command lightsOn = **new** LightOnConcreteCommand(light);

Command lightsOff = **new** LightOffConcreteCommand(light);

// switch on

control.setCommand(lightsOn);

control.pressButton();

// switch off

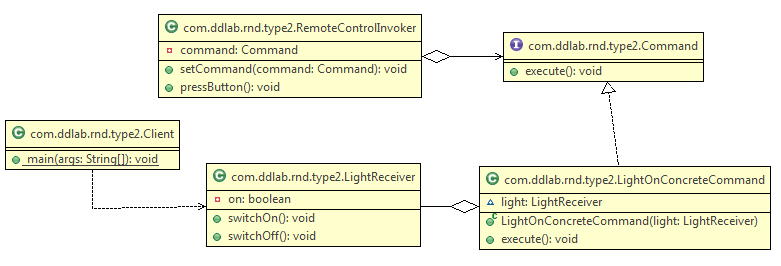
control.setCommand(lightsOff);

control.pressButton();

}

}

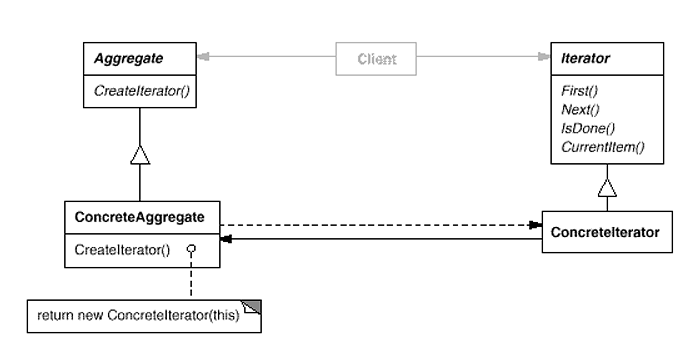
**UML Class Diagram** is given below.

****

Iterator Design Pattern

GOF : **Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation**.

# **Structure Diagram**

  
An example is given below.

import java.util.ArrayList;

import java.util.Iterator;

public class ArrayIterator implements Iterable {

**private Object[] elements;**

**private int size;**

**private int counter = 0;**

public ArrayIterator() {

**elements = new Object[20];**

}

**public void add( Object x ) {**

**elements[size++] = x;**

**}**

**@Override**

**public Iterator iterator() {**

**//reset the counter**

**//If you do not reset the counter, you will not be iterate once again**

**counter = 0;**

**return new MyIterator();**

**}**

**private class MyIterator implements Iterator {**

**@Override**

**public boolean hasNext() {**

**return counter < elements.length && elements[counter] != null ;**

**}**

**@Override**

**public Object next() {**

**return elements[counter++] ;**

**}**

@Override

public void remove() {

System.out.println("Don't want to delete item");

}

}

public static void main(String[] args) {

ArrayIterator arr = new ArrayIterator();

for( int i = 0 ; i < 10 ; i++ ) {

arr.add( new Integer(i));

}

Iterator itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next()); //0 1 2 3 4 5 6 7 8 9

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t" + itr.next()); //0 1 2 3 4 5 6 7 8 9

}

//In case of arraylist also, everytime, you get an iterator,

//the index is et to 0 so that it can be iterated.

ArrayList al = new ArrayList();

for( int i = 0 ; i < 10 ; i++ ) {

al.add( new Integer(i));

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next());

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next());

}

}

}

**What is an Aggregate Object ? An aggregate object is one which contains other objects. For example, an Airplane class would contain Engine, Wing, Tail, Crew objects. Aggregate is a collection of primitive and aggregate.**

The iterator pattern is a behavioral object design pattern. The iterator pattern allows for the traversal through the elements in a grouping of objects via a standardized interface. The code is given below.

public class Item {

String name;

float price;

public Item(String name, float price) {

this.name = name;

this.price = price;

}

public String toString() {

return name + ": $" + price;

}

}

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Menu {

List<Item> menuItems;

public Menu() {

menuItems = new ArrayList<Item>();

}

public void addItem(Item item) {

menuItems.add(item);

}

public Iterator<Item> iterator() {

return new MenuIterator();

}

class MenuIterator implements Iterator<Item> {

int currentIndex = 0;

@Override

public boolean hasNext() {

if (currentIndex >= menuItems.size()) {

return false;

} else {

return true;

}

}

@Override

public Item next() {

return menuItems.get(currentIndex++);

}

@Override

public void remove() {

menuItems.remove(--currentIndex);

}

}

}

import java.util.Iterator;

public class Demo {

public static void main(String[] args) {

Item i1 = new Item("spaghetti", 7.50f);

Item i2 = new Item("hamburger", 6.00f);

Item i3 = new Item("chicken sandwich", 6.50f);

Menu menu = new Menu();

menu.addItem(i1);

menu.addItem(i2);

menu.addItem(i3);

System.out.println("Displaying Menu:");

Iterator<Item> iterator = menu.iterator();

while (iterator.hasNext()) {

Item item = iterator.next();

System.out.println(item);

}

System.out.println("\nRemoving last item returned");

iterator.remove();

System.out.println("\nDisplaying Menu:");

iterator = menu.iterator();

while (iterator.hasNext()) {

Item item = iterator.next();

System.out.println(item);

}

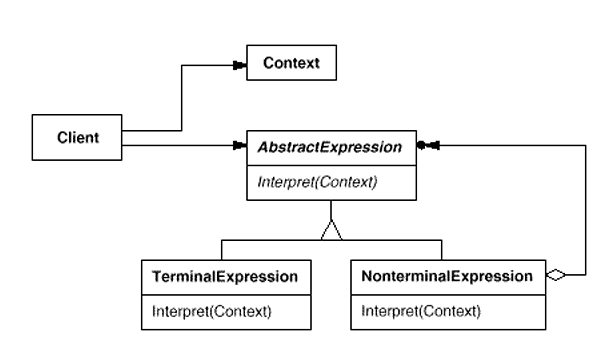
}

}

Interpreter Pattern

GOF : **Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.**

# **Structure Diagram**



Example : Evaluate an expression like “5 + 10”.

**Numbers.java**

**public** **class** Numbers **implements** Expression {

**private** **int** num;

**public** Numbers( **int** num ) {

**this**.num = num;

}

@Override

**public** **int** calc() {

**return** **this**.num;

}

}

The code is given below.

**Evaluator.java  
public** **class** Evaluator {

**Add.java**

**public** **class** Add **implements** Expression {

**private** Expression lhs;

**private** Expression rhs;

**public** Add(Expression lhs , Expression rhs) {

**this**.lhs = lhs;

**this**.rhs = rhs;

}

@Override

**public** **int** calc() {

**return** **this**.lhs.calc()+**this**.rhs.calc();

}

}

**Expression.java**

**public** **interface** Expression {

**int** calc();

}

**public** **int** evaluate( String statement ) {

String[] exps = statement.split(" ");

**int** leftOperand = Integer.*parseInt*( exps[0]);

**int** rightOperand = Integer.*parseInt*( exps[2]);

String operation = exps[1];

**return** **new** Add( **new** Numbers(leftOperand) , **new** Numbers(rightOperand) ).calc();

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

String statement = "5 + 10";

Evaluator evalutator = **new** Evaluator();

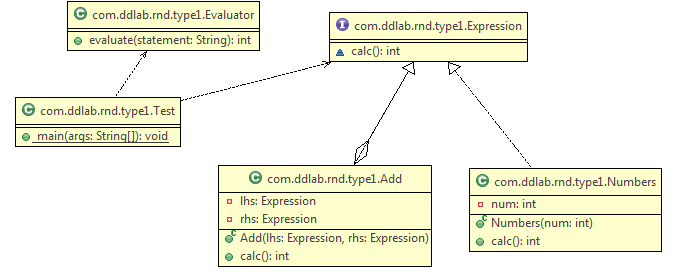
**int** result = evalutator.evaluate(statement);

System.***out***.println("Result :::"+result);

}

}

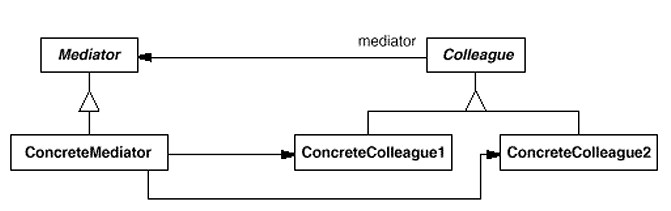
UML diagram is given below.

****

Mediator Pattern

GOF: **Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently**.

# **Structure Diagram**

  
**Java Code**

**Mediator.java**

//Mediator interface  
public interface Mediator {  
 public void send(String message, Colleague colleague);  
}

**Colleague.java**  
//Colleage interface  
public abstract class Colleague {  
 private Mediator mediator;  
  
 public Colleague(Mediator m) {  
 mediator = m;  
 }  
  
 //send a message via the mediator  
 public void send(String message) {  
 mediator.send(message, this);  
 }  
  
 //get access to the mediator  
 public Mediator getMediator() {  
 return mediator;  
 }  
 public abstract void receive(String message);  
}

**ConcreteColleague.java**public class ConcreteColleague extends Colleague {  
 public ConcreteColleague(Mediator mediator) {  
 super(mediator);  
 }  
  
 public void receive(String message) {  
 System.*out*.println("Colleague Received: " + message);  
 }  
}

**MobileColleague**

public class MobileColleague extends Colleague {  
 public MobileColleague(Mediator mediator) {  
 super(mediator);  
 }  
  
 public void receive(String message) {  
 System.*out*.println("Mobile Received: " + message);  
 }  
}

**ApplicationMediator.java**

import java.util.ArrayList;  
public class ApplicationMediator implements Mediator {  
  
 private ArrayList<Colleague> colleagues;  
  
 public ApplicationMediator() {  
 colleagues = new ArrayList<Colleague>();  
 }  
  
 public void addColleague(Colleague colleague) {  
 colleagues.add(colleague);  
 }  
  
 public void send(String message, Colleague originator) {  
 //let all other screens know that this screen has changed  
 for (Colleague colleague : colleagues) {  
 //don't tell ourselves  
 if (colleague != originator) {  
 colleague.receive(message);  
 }  
 }  
 }  
}

**Client.java**

public class Client {  
 public static void main(String[] args) {  
 ApplicationMediator mediator = new ApplicationMediator();  
  
 Colleague desktop = new ConcreteColleague(mediator);  
  
 Colleague mobile = new MobileColleague(mediator);  
  
 mediator.addColleague(desktop);  
 mediator.addColleague(mobile);  
  
  
 desktop.send("Hello World");  
 mobile.send("Hello");  
 }  
}

Create an “intermediary” that decouples “senders” from “receivers”

Producers are coupled only to the Mediator , Consumers are coupled only to the Mediator

The Mediator arbitrates the storing and retrieving of messages

http://en.wikipedia.org/wiki/Mediator\_pattern

The mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior. You can consider the example on Cosumer and Producer in multi threading.

public class Player1 extends Thread

{

private Object monitor = null;

private StatusHolder status = null;

public Player1( Object monitor , StatusHolder holder )

{

this.monitor = monitor;

this.status = holder;

}

public void run()

{

synchronized( monitor )

{

while( true )

{

try

{

System.out.println("Status for Player 1---->"+status.hasCompleted);

if( status.hasCompleted == false )

{

monitor.wait();

}

System.out.println("Player 1 is playing the game ...");

Thread.sleep(1000);

status.hasCompleted = false;

System.out.println("Player 1 has completed the turn and going to notify");

System.out.println("-----------------------END for Player 1-------------------------");

monitor.notify();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

}

}

public class Player2 extends Thread

{

private Object monitor = null;

private StatusHolder status = null;

public Player2( Object monitor , StatusHolder holder )

{

this.monitor = monitor;

this.status = holder;

}

public void run()

{

synchronized( monitor )

{

while( true )

{

try

{

System.out.println("Status for Player 2---->"+status.hasCompleted);

if( status.hasCompleted == true )

{

monitor.wait();

}

System.out.println("Player 2 is playing the game ...");

Thread.sleep(1000);

System.out.println("Player 2 has completed the turn and going to notify");

status.hasCompleted = true;

System.out.println("-----------------------END for Player 2-------------------------");

monitor.notify();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

}

}

public class StatusHolder //Mediator Design Pattern

{

public boolean hasCompleted = false;

}

public class TestGame {

public static void main(String[] args)

{

Object monitor = new Object();

StatusHolder holder = new StatusHolder();

new Player1(monitor, holder).start();

new Player2(monitor, holder).start();

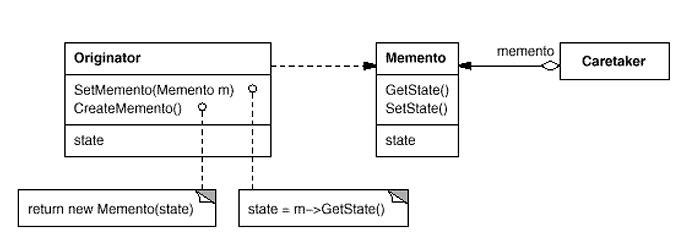
}

}

Memento Pattern

GOF : **Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.**

**Structure Diagram**

  
Java Code Example is given below.

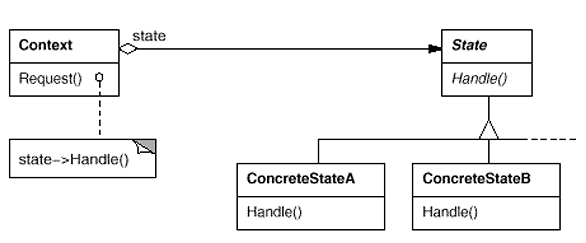
class Originator {  
 private String state;  
 // The class could also contain additional data that is not part of the  
 // state saved in the memento..  
  
 public void set(String state) {  
 System.*out*.println("Originator: Setting state to " + state);  
 this.state = state;  
 }  
  
 public Memento saveToMemento() {  
 System.*out*.println("Originator: Saving to Memento.");  
 return new Memento(this.state);  
 }  
  
 public void restoreFromMemento(Memento memento) {  
 this.state = memento.getSavedState();  
 System.*out*.println("Originator: State after restoring from Memento: " + state);  
 }  
  
 **public static class Memento {  
 private final String state;  
  
 public Memento(String stateToSave) {  
 state = stateToSave;  
 }  
  
 public String getSavedState() {  
 return state;  
 }  
 }**}

import java.util.ArrayList;  
import java.util.List;  
  
class Caretaker {  
 public static void main(String[] args) {  
 List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();  
   
 Originator originator = new Originator();  
 originator.set("State1");  
 originator.set("State2");  
 savedStates.add(originator.saveToMemento());  
 originator.set("State3");  
 // We can request multiple mementos, and choose which one to roll back to.  
 savedStates.add(originator.saveToMemento());  
 originator.set("State4");  
   
 originator.restoreFromMemento(savedStates.get(1));   
 }  
}

State Pattern

GOF : **Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.**

**Structure Diagram**

  
**Java Code Example**

**State.java**

public interface State {  
 public void doAction(Context context);  
}

**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";  
 }  
}

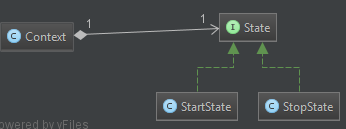
**Context.java**

public class Context {  
 private State state;  
  
 public Context() {  
 state = null;  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 }  
  
 public State getState() {  
 return state;  
 }  
}

**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());  
 }  
}

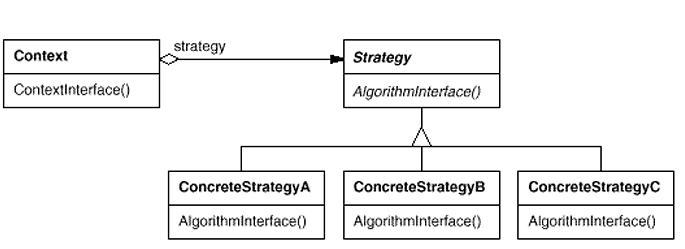
UML Diagram for the above is given below.



Strategy Pattern

GOF : **Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it**.

# **Structure Diagram**

  
http://en.wikipedia.org/wiki/Strategy\_pattern

In computer programming, the strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime. Formally speaking, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The example is given below.

// The classes that implement a concrete strategy should implement this.

// The Context class uses this to call the concrete strategy.

interface IStrategy {

int execute(int a, int b);

}

// Implements the algorithm using the strategy interface

class ConcreteStrategyAdd implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategyAdd's execute()");

return a + b; // Do an addition with a and b

}

}

class ConcreteStrategySubtract implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategySubtract's execute()");

return a - b; // Do a subtraction with a and b

}

}

class ConcreteStrategyMultiply implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategyMultiply's execute()");

return a \* b; // Do a multiplication with a and b

}

}

// Configured with a ConcreteStrategy object and maintains a reference to a Strategy object

class Context {

private IStrategy strategy;

// Constructor

public Context(IStrategy strategy) {

this.strategy = strategy;

}

public int executeStrategy(int a, int b) {

return strategy.execute(a, b);

}

}

// Test application

class StrategyExample {

public static void main(String[] args) {

Context context;

// Three contexts following different strategies

context = new Context(new ConcreteStrategyAdd());

int resultA = context.executeStrategy(3,4);

context = new Context(new ConcreteStrategySubtract());

int resultB = context.executeStrategy(3,4);

context = new Context(new ConcreteStrategyMultiply());

int resultC = context.executeStrategy(3,4);

System.out.println("Result A : " + resultA );

System.out.println("Result B : " + resultB );

System.out.println("Result C : " + resultC );

}

}

Another example is given below.

**CompressionStrategy.java**  
import java.io.File;  
import java.util.ArrayList;  
  
//Strategy Interface  
public interface CompressionStrategy {  
 public void compressFiles(ArrayList<File> files);  
}

**CompressionContext.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class CompressionContext {  
 private CompressionStrategy strategy;  
  
 public CompressionContext(CompressionStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 //use the strategy  
 public void createArchive(ArrayList<File> files) {  
 strategy.compressFiles(files);  
 }  
}

**RarCompressionStrategy.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class RarCompressionStrategy implements CompressionStrategy {  
 public void compressFiles(ArrayList<File> files) {  
 //using RAR approach  
 System.*out*.println("Compressing files using ZIP strategy ...");  
 }  
}

**ZipCompressionStrategy.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class ZipCompressionStrategy implements CompressionStrategy {  
 public void compressFiles(ArrayList<File> files) {  
 //using ZIP approach  
 System.*out*.println("Compressing files using ZIP strategy ...");  
 }  
}

**Client.java**

import java.io.File;  
import java.util.ArrayList;  
import java.util.List;  
  
public class Client {  
 public static void main(String[] args) {  
 CompressionContext ctx = new CompressionContext(new ZipCompressionStrategy());  
 //we could assume context is already set by preferences  
  
 //get a list of files...  
 ArrayList<File> files = new ArrayList<File>();  
 File file1 = new File("a.doc");  
 File file2 = new File("b.doc");  
 files.add(file1);  
 files.add(file2);  
  
 ctx.createArchive(files);  
 }  
}

UML Diagram

