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());

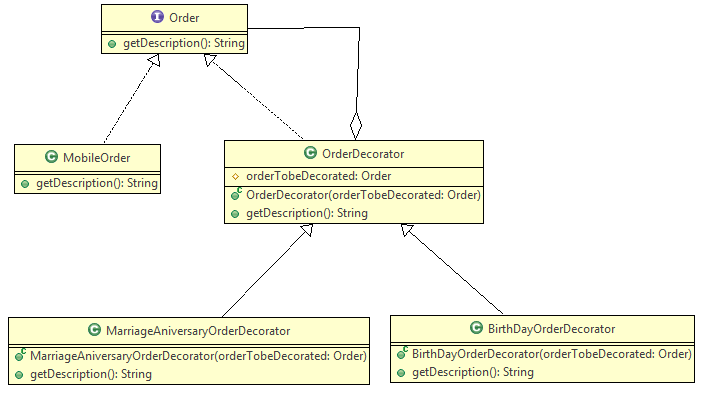
}

}

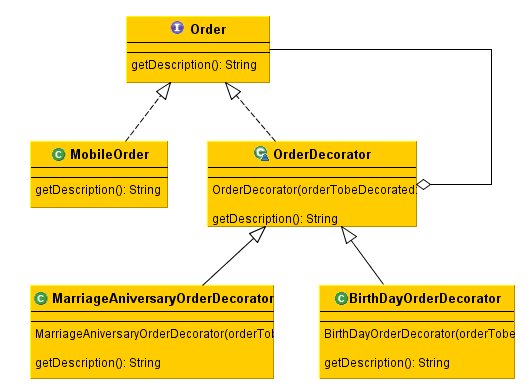
**Output**

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());

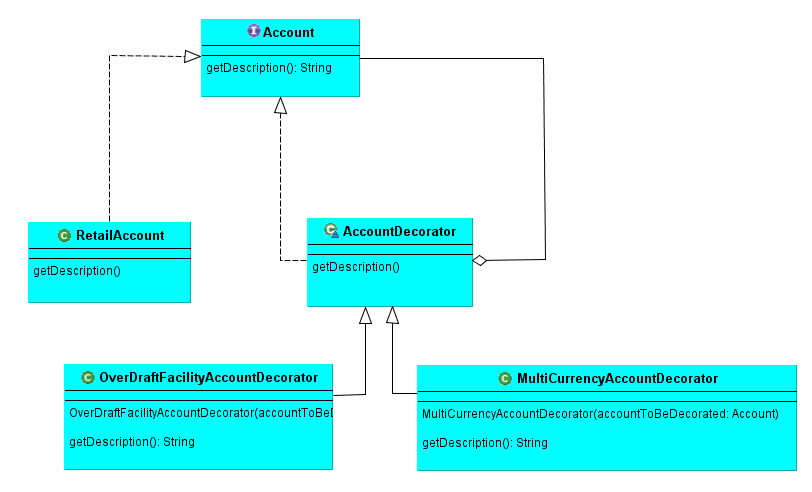
//Output -> Retail account, including overdraft facility

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);  
 }  
}

