

[JAVA TUTORIAL](#)[#INDEX POSTS](#)[#INTERVIEW QUESTIONS](#)[RESOURCES](#)[HIRE ME](#)[DOWNLOAD ANDROID APP](#)[CONTRIBUTE](#)**Subscribe to Download Java Design Patterns eBook****DOWNLOAD NOW**[HOME](#) » [JAVA](#) » [DESIGN PATTERNS](#) » [DECORATOR DESIGN PATTERN IN JAVA EXAMPLE](#)

Decorator Design Pattern in Java Example

APRIL 2, 2018 BY [PANKAJ](#) — [22 COMMENTS](#)

Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as [Adapter Pattern](#), [Bridge Pattern](#), [Composite Pattern](#)) and uses abstract classes or interface with [composition](#) to implement.

Table of Contents [\[hide\]](#)

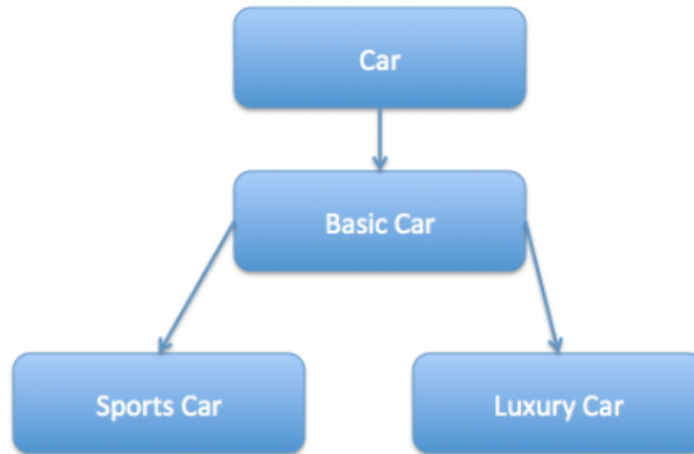
- [1 Decorator Design Pattern](#)
- [2 Decorator Design Pattern – Class Diagram](#)
 - [2.1 Decorator Design Pattern Test Program](#)
 - [2.2 Decorator Design Pattern – Important Points](#)

Decorator Design Pattern

We use [inheritance](#) or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality or remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The

implementation hierarchy will look like below image.



But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

We need to have following types to implement decorator design pattern.

1. **Component Interface** – The interface or **abstract class** defining the methods that will be implemented. In our case Car will be the component interface.

```
package com.journaldev.design.decorator;  
  
public interface Car {  
  
    public void assemble();  
}
```

2. **Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.

```
package com.journaldev.design.decorator;  
  
public class BasicCar implements Car {  
  
    @Override  
    public void assemble() {  
        System.out.print("Basic Car.");  
    }  
}
```

```
}
```

```
}
```

3. **Decorator** – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

```
package com.journaldev.design.decorator;

public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }

}
```

4. **Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

```
package com.journaldev.design.decorator;

public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }

}
```

```

package com.journaldev.design.decorator;

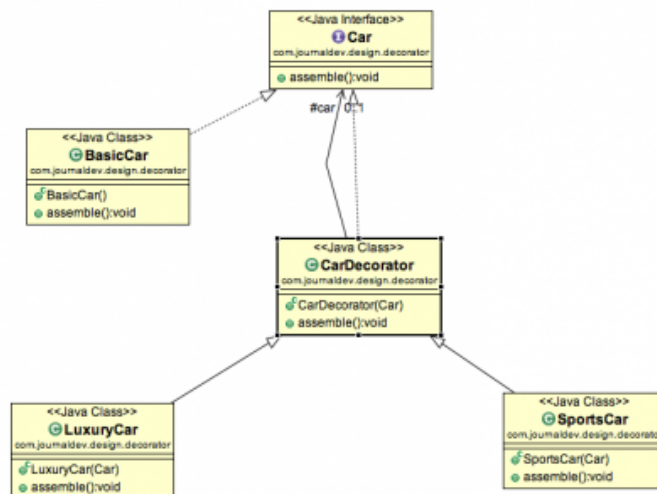
public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}

```

Decorator Design Pattern – Class Diagram



Decorator Design Pattern Test Program

```

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;
import com.journaldev.design.decorator.Car;
import com.journaldev.design.decorator.LuxuryCar;
import com.journaldev.design.decorator.SportsCar;

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();
    }

}
```

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

```
Basic Car. Adding features of Sports Car.
```

```
*****
```

```
Basic Car. Adding features of Luxury Car. Adding features of Sports Car.
```

Decorator Design Pattern – Important Points

- Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
- The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
- Decorator pattern is used a lot in **Java IO** classes, such as **FileReader**, **BufferedReader** etc.

« PREVIOUS

Composite Design Pattern in Java

NEXT »

Facade Design Pattern in Java

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on [Google Plus](#), [Facebook](#) or [Twitter](#). I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on [Youtube](#).

FILED UNDER: [DESIGN PATTERNS](#)

Comments

Mark Schumacher says

MAY 26, 2018 AT 6:37 AM

Ah ha This is a Junit rule .

Statement.apply((Statement) base)

.

ahh :(..... I havent been outside for a while

[Reply](#)**Arun Singh says**

MAY 30, 2017 AT 12:28 PM

How to get the features of Luxury Car separately if I need them for another car at my home.

I assume the same scenario is for Dosa where you can get plain dosa, dosa with masala and dosa with chutney.

So masala and chutney classes extend DosaDecorator. PlainDosa class and DosaDecorator class implements Dosa interface.

Now what if i want extra chutney (means only chutney separately) ?

Thanks in advance.

[Reply](#)**Harsh Gupta says**

JANUARY 1, 2018 AT 6:26 PM

Hi Arun,

Looks like with Decorator DP it is not possible to get extra chutney. You can't ask for more aloo or chutney.

Actually chutney here is not a separate class, it is dependent on dosa interface. You will have to use some other DP.

But you brought the nice question over here. Were you able to figure out the answer to your problem?

[Reply](#)**Romit Chhabra says**

MARCH 1, 2018 AT 5:40 AM

It can be done by changing implementation of assemble method in CarDecorator a bit.

```
@Override
public void assemble() {
    if(c != null) {
        c.assemble();
    }
}
```

And creating an object of Chutney as

```
Dosa chutney = new Chutney(null);
```

```
chutney.assemble();
```

[Reply](#)

Arun Singh says

MAY 30, 2017 AT 11:38 AM

How to print only Luxury Car?

Thanks

[Reply](#)

DEBARATI MAJUMDER says

AUGUST 19, 2017 AT 5:34 AM

```
new LuxuryCar(new BasicCar());
```

[Reply](#)

Agustin Labaronnie says

APRIL 10, 2017 AT 12:39 PM

Hello. I used your tutorial to implement a Decorator, but found an error:

sportsLuxuryCar.assemble(); won't execute the assemble() method for every Decorator used when creating the instance. So the output of sportsLuxuryCar.assemble(); won't be:

"Basic Car. Adding features of Luxury Car. Adding features of Sports Car." but "Basic Car. Adding features of Sports Car." instead.

I think you should correct that in the tutorial, because the chain execution of the assemble() method is useful and should be invoked in constructors in order to work.

Thank you!

[Reply](#)

Masoud says

NOVEMBER 17, 2016 AT 7:23 AM

I think CarDecorator should be an abstract class in order for it to confirm to decorator design pattern ,

[Reply](#)

Bal says

APRIL 24, 2016 AT 7:59 AM

The relationship between a car and the decorator is inheritance, doesn't sound quite right to call a CarDecorator "is a" Car.

[Reply](#)**Pankaj says**

JUNE 10, 2016 AT 11:07 PM

CarDecorator has to implement Car interface to provide methods to create Car objects. If you won't implement Car interface then below code will start giving compile time error.

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

[Reply](#)**Pankaj says**

SEPTEMBER 5, 2016 AT 7:28 AM

Here inheritance is not used in its usual type "is a ". Here inheritance provides common supertype for car decorator and basic car.

[Reply](#)**prakash says**

FEBRUARY 24, 2016 AT 8:03 AM

Pankaj sir

multiple design pattern using create Framework?

example

login Framework

[Reply](#)**Pankaj says**

JUNE 10, 2016 AT 11:43 PM

Not sure what you meant here.

[Reply](#)

ildar says

DECEMBER 10, 2015 AT 12:48 AM

I believe you should put "super.assemble();" instead of "car.assemble();" in concrete decorators. But still your tutorial is great! Thank you very much!

[Reply](#)**Pankaj says**

JUNE 10, 2016 AT 11:44 PM

Thanks for the input, yes you are right. I have corrected it.

[Reply](#)**Muhammad Gelbana says**

JUNE 10, 2015 AT 4:11 AM

You said: "Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage"

But then you still created 2 classes for each type of car (LuxuryCar & SportsCar). So the this pattern didn't solve the problem of having ten different kind of cars.

Or what did I get wrong here ?

Thank you.

[Reply](#)**Brian Laughlin says**

AUGUST 30, 2015 AT 9:52 AM

I believe you will need to use another pattern like the Factory pattern. One drawback to the decorator pattern is that it can add a lot of smaller classes and increase the complexity of the code.

[Reply](#)**Pankaj says**

JUNE 11, 2016 AT 12:35 AM

Yes, this is the drawback of Decorator pattern, you will have a lot of similar classes.

[Reply](#)**Ali Rekmani says**

SEPTEMBER 13, 2016 AT 8:44 PM

The example with 2 classes helps to explain the concept, but in real life, in grosso modo, through n different decorator classes you can define n! different subclasses, it's worth it

[Reply](#)

john says

DECEMBER 18, 2013 AT 11:48 AM

I want to ask you a question. How did you draw a class diagram?
which tool? Could you show some useful UML plugins for eclipse ?
Thank you.

[Reply](#)

Pankaj says

JUNE 11, 2016 AT 12:42 AM

I use ObjectAid UML Explorer for class diagram. It's a simple tool and quick to install.

[Reply](#)

Amit says

JULY 14, 2013 AT 9:57 AM

Thanks for the nice tutorial.

[Reply](#)

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *



Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP



DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- > [Singleton](#)
- > [Factory](#)
- > [Abstract Factory](#)
- > [Builder](#)
- > [Prototype](#)

Structural Design Patterns

- > [Adapter](#)
- > [Composite](#)
- > [Proxy](#)
- > [Flyweight](#)
- > [Facade](#)
- > [Bridge](#)
- > [Decorator](#)

Behavioral Design Patterns

- > [Template Method](#)
- > [Mediator](#)
- > [Chain of Responsibility](#)

- > [Observer](#)
- > [Strategy](#)
- > [Command](#)
- > [State](#)
- > [Visitor](#)
- > [Interpreter](#)
- > [Iterator](#)
- > [Memento](#)

Miscellaneous Design Patterns

- > [Dependency Injection](#)
- > [Thread Safety in Java Singleton](#)

RECOMMENDED TUTORIALS

Java Tutorials

- > [Java IO](#)
- > [Java Regular Expressions](#)
- > [Multithreading in Java](#)
- > [Java Logging](#)
- > [Java Annotations](#)
- > [Java XML](#)
- > [Collections in Java](#)
- > [Java Generics](#)
- > [Exception Handling in Java](#)
- > [Java Reflection](#)
- > [Java Design Patterns](#)
- > [JDBC Tutorial](#)

Java EE Tutorials

- > [Servlet JSP Tutorial](#)
- > [Struts2 Tutorial](#)
- > [Spring Tutorial](#)
- > [Hibernate Tutorial](#)
- > [Primefaces Tutorial](#)
- > [Apache Axis 2](#)
- > [JAX-RS](#)
- > [Memcached Tutorial](#)

Oracle Database Tools

2018 New Luxury Cars

Top 5 Hybrid Cars

20 Best Ford Cars

Cars Under 500

Pre-Owned Car Dealers

Best Hybrid Cars

New Hybrid & Electric Car

© 2018 · Privacy Policy · Don't copy, it's Bad Karma · P

