STS #INTERVIEW QUESTIONS RESOURCES

Instantly Search Tutorials...

ARE HERE: HOME » JAVA » DESIGN PATTERNS » DECORATOR DESIGN PATTERN IN JAVA MPLE

Decorator Design Pattern in Java Example

ANKAJ — 30 COMMENTS

Patterns

- Adapter
- Composite
- Proxy
- Flyweight
- Facade

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.

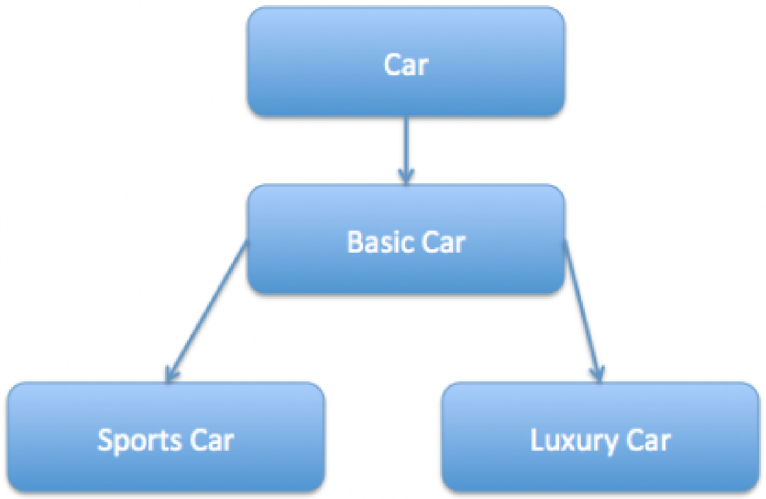
Bridge

Table of Contents [hide]

Decorator Design Pattern

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

Suppose we want to implement different kinds of cars – we can create nterface Car to define the assemble method and then we can have a Basic ar, further more we can extend it to Sports car and Luxury Car. The nplementation hierarchy will look like below image.



Tutorials

✚ Java Tutorials

✚ Java EE Tutorials

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

Ve 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.");
        }
}
```

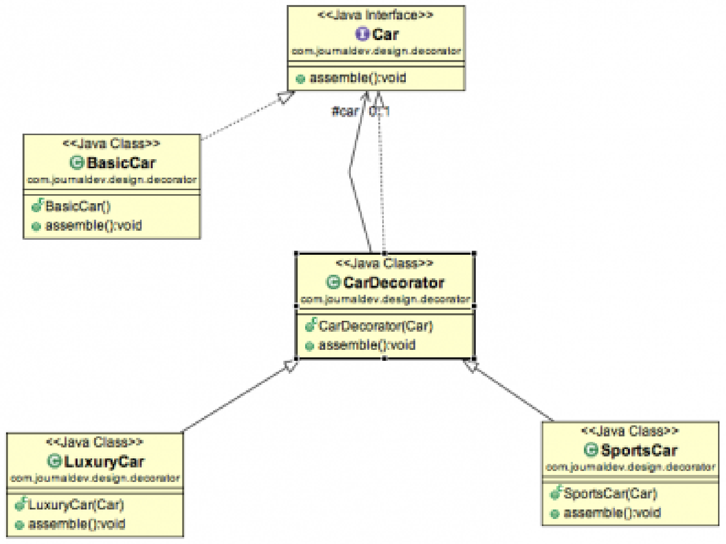
```
package com.journaldev.design.decorator;

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.");
        }
}
```

ecorator Design Pattern – Class iagram



Decorator Design Pattern Test Program

Copy

```
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();
        }

}
```

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

Jutput of above test program is:

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

Jecorator 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.

\dapter Design Pattern in Java

Facade Design Pattern in Java

Composite Design Pattern in Java

Bridge Patter

Factory Design Pattern in Java

DAO Design Pattern

Abstract Factory Design Pattern in Java

Java Patt Tutc

REVIOUS
nposite 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**.

ILED UNDER: DESIGN PATTERNS

Comments

alex says

MARCH 2, 2019 AT 5:56 AM

Great concise and very clear explanation. Thanks!

Reply

Andre says

FEBRUARY 12, 2019 AT 8:58 AM

This is a great example, thank you.
If I may suggest, I believe it would be more didactic if you use Options instead of Cars for the Decorator related classes, this way it's easier to understand that the final Car will be the assemble of different options (or features).

Reply

Radhikabahen Patel says

DECEMBER 21, 2018 AT 7:44 PM

Another example

https://github.com/radhikapatel4391/CS680/tree/master/Homework3/src/hw3

Very nice tutorial point to point and simple easy to understand.

Reply

Radhikabahen Patel says

DECEMBER 21, 2018 AT 7:42 PM

DVDPlayer example..

https://github.com/radhikapatel4391/CS680/tree/master/Homework3/src/hw3

Nice blog…

Reply

Divyansh Anand says

OCTOBER 21, 2018 AT 12:17 PM

Very helpful information, I’m glad you shared it with people like us who are learning it

Reply

Farukh says

SEPTEMBER 30, 2018 AT 9:13 PM

Hello,everyone! I have Monster(main class) and some Concrete classes(, like Troll,Vampire). Also I have MonsterDecorators,like Club,Sword and etc.
EXAMPLE:
Monster troll = new Troll();
troll.getAttackPower() // 30
troll.attack(); // The troll tries to grab you!
troll.fleeBattle(); // The troll shrieks in horror and runs away!
–>change the behavior of the simple troll by adding a decorator
troll = new ClubDecorator(troll);
troll.getAttackPower() // 42
troll.attack(); // The troll tries to grab you! The troll swings at you with a club!
troll.fleeBattle(); // The troll shrieks in horror and runs away! And loses his club while running!
so after fleeBattle method, how troll can be just Troll, without any decorators?

Reply

Name says

NOVEMBER 12, 2018 AT 12:57 AM

Not sure if it will work but it's worth a try:
toll = new Troll();

Reply

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

arpit says

SEPTEMBER 6, 2018 AT 9:51 AM

@ArunSingh You asked solution for extra chutney. If you want double chutney with plain dosa then you use below code

Dosa dosa = new PlainDosa();
dosa = new Chutney(dosa);
dosa = new Chutney(dosa);

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

Agustín 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(); wont' 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 image 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

nt Policy: Please submit comments to add value to the post. Comments like "Thank You" and ne Post" will be not published. If you want to post code then wrap them inside <pre> tags. For : **<pre>class Foo { }</pre>**.

ant to post XML content, then please escape < with &lt; and > with &gt; otherwise they will not be roperly.

Leave a Reply

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

Comment

lame *

:mail *

POST COMMENT