

Decorator Design Pattern in Java

📅 Last Updated: August 22, 2021 👤 By: Lokesh Gupta 📁 Structural Patterns 💡 Design Patterns

In software engineering, **decorator design pattern** is used to add additional features or behaviors to a particular instance of a class, while not modifying the other instances of same class. Decorators provide a flexible alternative to sub-classing for extending functionality. Please note that the description above implies that decorating an object changes its behavior but not its interface.

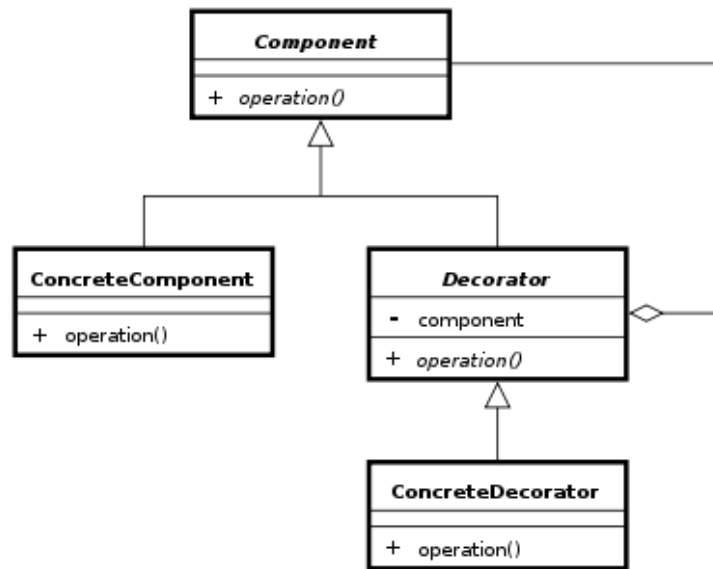
This pattern is very important to understand because once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities without making any code changes to the underlying classes. Interesting, isn't it?? Also this pattern is really useful and commonly faced [java interview question](#) on [design patterns](#).

In this post we will discuss on following points:

- Design participants
- Sample problem statement
- Proposed solution using Decorator design pattern
- Common interview questions
- Some practical usages

Design participants

A typical diagram of decorator pattern looks like this.



Decorator design pattern participants

Following are the participants of the Decorator Design pattern:

- **Component** – this is the wrapper which can have additional responsibilities associated with it at runtime.
- **Concrete component**– is the original object to which the additional responsibilities are added in program.
- **Decorator**-this is an abstract class which contains a reference to the component object and also implements the component interface.
- **Concrete decorator**-they extend the decorator and builds additional functionality on top of the Component class.

If you closely read between the lines then you will understand that it works like this:

You have an instance, and you put another instance inside of it. They both support the same (or similar) interfaces. The one on the outside is a "decorator." You use the one on the outside. It either masks, changes, or pass-throughs the methods of the instance inside of it.

I always prefer to learn things by example. So, let's have one problem and let's solve it.

Problem statement

Let's have a common usecase where we have to show all user created reports to admin. Now, these reports can fall into these categories.

- Client reports
- Support reports

Both of these reports **must have their first column as a link** to original report and they should have **different coloring**. Possibly they should be opened in **different popup window sizes**. These are just few things, a lot can come in reality.

A **possible approach can be to extend the Report object**, and then have two separate classes ClientReport and SupportReport. In both methods, define private methods e.g. *makeAnchorLink()*, *designPopup()*, *applyColor()* and so on. Now call these methods in some getter method for first column data cell.

It will work if you stop here and do not modify the system anymore. But, let's say after few days **you are told to apply different background colors** for both kinds of report. Now you have only one way that define a method *colorBackground()* in both classes and call appropriately. Problem becomes worse when you have more number of reports in your system in future.

Above solution is a clear violation of **Open/Closed principle** mentioned in [SMART principles for class design](#).

Proposed solution using Decorator design pattern

Above problem is a perfect candidate for decorator pattern. Remember when we have an object that requires the extension but by design that is not suitable, go for decoration.

Above problem can be solved easily by following class diagram. Here I am using only for Support reports. A similar class hierarchy can be build for client reports also.

Decorator Pattern Solution

If we have implemented our solution like this, anytime in future we can add additional decoration without modifying existing class hierarchy. That is the ultimate goal we had in starting of this post, right?

Sourcecode Listings

Let's see above class's source code to know how actually it looks like.

Report.java

```
public interface Report {  
    public Object[][] getReportData(final String reportId);  
    public String getFirstColumnData();  
}
```

SupportReport.java

```
public class SupportReport implements Report {  
  
    @Override  
    public Object[][] getReportData(String reportId) {  
        return null;  
    }  
  
    @Override  
    public String getFirstColumnData() {  
        return "Support data";  
    }  
}
```

ColumDecorator.java

```
public abstract class ColumDecorator implements Report  
{  
    private Report decoratedReport;  
  
    public ColumDecorator(Report report){  
        this.decoratedReport = report;  
    }  
  
    public String getFirstColumnData() {  
        return decoratedReport.getFirstColumnData();  
    }  
  
    @Override  
    public Object[][] getReportData(String reportId) {  
        return decoratedReport.getReportData(reportId);  
    }  
}
```

SupportLinkDecorator.java

```
public class SupportLinkDecorator extends ColumDecorator{
```

```

public SupportLinkDecorator(Report report) {
    super(report);
}

public String getFirstColumnData() {
    return addMoreInfo (super.getFirstColumnData()) ;
}

private String addMoreInfo(String data){
    return data  + " - support link - ";
}
}

```

SupportPopupDecorator.java

```

public class SupportPopupDecorator extends ColumDecorator{

    public SupportPopupDecorator(Report report) {
        super(report);
    }

    public String getFirstColumnData() {
        return addPopup (super.getFirstColumnData()) ;
    }

    private String addPopup(String data){
        return data  + " - support popup - ";
    }
}

```

DecoratorPatternTest.java

```

public class DecoratorPatternTest {
    public static void main(String[] args) {


        //ClientPopupDecorator popupDecorated = new ClientPopupDecorator(new ClientL
        //System.out.println(popupDecorated.getFirstColumnData());

        SupportPopupDecorator supportPopupDecorated = new SupportPopupDecorator(new
        System.out.println(supportPopupDecorated.getFirstColumnData());
    }
}

```

Output:

Support data - support link - - support popup -



To download the complete source code and UML diagram, follow the download link in the end of post.

Interview questions on decorator pattern

A) How to decide when to use decorator pattern?

If we drill down more on the concept, we find that Decorator Design Pattern has several requirement indicators to suggest that it is potential solution e.g.

- We have an object that requires the extension e. a window control that requires additional "optional" features like scrollbars, titlebar and statusbar.
- Several objects that support the extension by "decoration". Usually, those objects share a common interface, traits, or superclass, and sometimes, additional, intermediate super-classes .
- The decorated object (class or prototype instantiation), and the decorator objects have one or several common features. In order to ensure that functionality, the decorated object & the decorators have a common interface, traits, or class inheritance.

B) Difference between decorator pattern and Adapter pattern

No. AdapterPattern is used to convert the interface of an object into something else. DecoratorPattern is used to extend the functionality of an object while maintaining its interface. Both of these are probably sometimes called Wrapper Pattern since both of them do "wrap" an object.

C) Difference between a DecoratorPattern and Subclassing

The difference between a DecoratorPattern and subclassing is that In subclassing you can decorate any class that implements an interface "with a single class". Say I wanted

to give myself a `java.util.Map` that printed a message whenever I added or removed a key. If I only ever actually used `java.util.HashMap` I could just create `PrintingMap?` as a subclass of `HashMap` and override `put` & `remove`. But if I want to create a printing version of `TreeMap` then I either create `PrintingTreeMap?` (which has almost identical code to `PrintingMap?`

Common usage of decorator pattern:

- 1) Java iO library classes e.g. `BufferedInputStream bs = new BufferedInputStream(new FileInputStream(new File("File1.txt")));`
- 2) In decorator column data in `display-tag jsp` library e.g.

```
<display:table name="reportsViewResultTable" class="demoClass" id="reportsQueryVie
  <display:column title="Report Id" sortable="true" property="reportDisplayId" dec
</display:table>
```

- 3) Decorators are used in **sitemesh**, to give a consistent UI experience.

Sourcecode Download

Happy Learning !!

Was this post helpful?

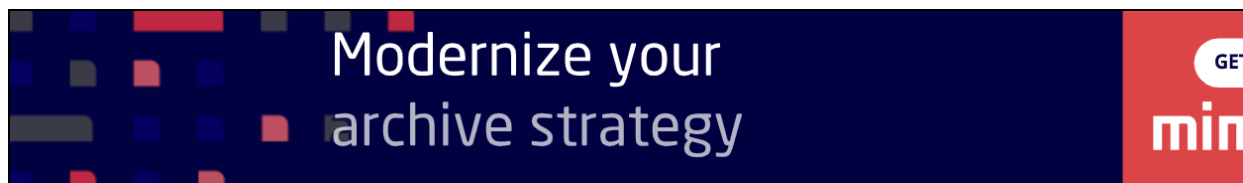
Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. [Adapter Design Pattern in Java](#)
2. [Bridge Design Pattern](#)
3. [Composite Design Pattern](#)
4. [Facade Design Pattern](#)
5. [Flyweight Design Pattern](#)
6. [Proxy Design Pattern](#)
7. [Prototype design pattern in Java](#)
8. [Chain of Responsibility Design Pattern](#)
9. [Template Method Design Pattern](#)
0. [Command Design Pattern](#)



Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

8 thoughts on “Decorator Design Pattern in Java”

Gayan

May 29, 2014 at 11:33 am

Thanks Lokesh.. Simple and understandable... And great comment also “Nothing feels better than a sense of being a part of someone's success” Best of luck to you..

[Reply](#)

Sabarish Chandrasekharan

March 11, 2014 at 3:57 am

Will AOP proxy be a candidate of Decorator pattern? As it to decorates the existing object with a added functionality

[Reply](#)**Lokesh Gupta**[March 11, 2014 at 9:04 am](#)

Absolutely yes. It's correct.

[Reply](#)**Jitendra Parida**[December 17, 2013 at 3:17 pm](#)

Nice example with simple coding style.
thanks

[Reply](#)**Ankur**[December 11, 2013 at 6:34 am](#)

Stumbled upon your blog sometime back ..Must say it is one of the better blogs clearing java concepts available on web . Really helpful.Hope u keep them coming 😊

[Reply](#)

Muhammad Mazhar Hassan

November 19, 2013 at 4:50 pm

Nice one Lokesh

[Reply](#)

Shankar Morwal

September 30, 2013 at 10:02 pm

Nice article sir. I enjoy reading your blog. I want to thank you, recently i cracked a dream interview with help of this blog.

[Reply](#)

Lokesh Gupta

September 30, 2013 at 11:04 pm

Great. Congratulations buddy. Nothing feels better than a sense of being a part of someone's success.

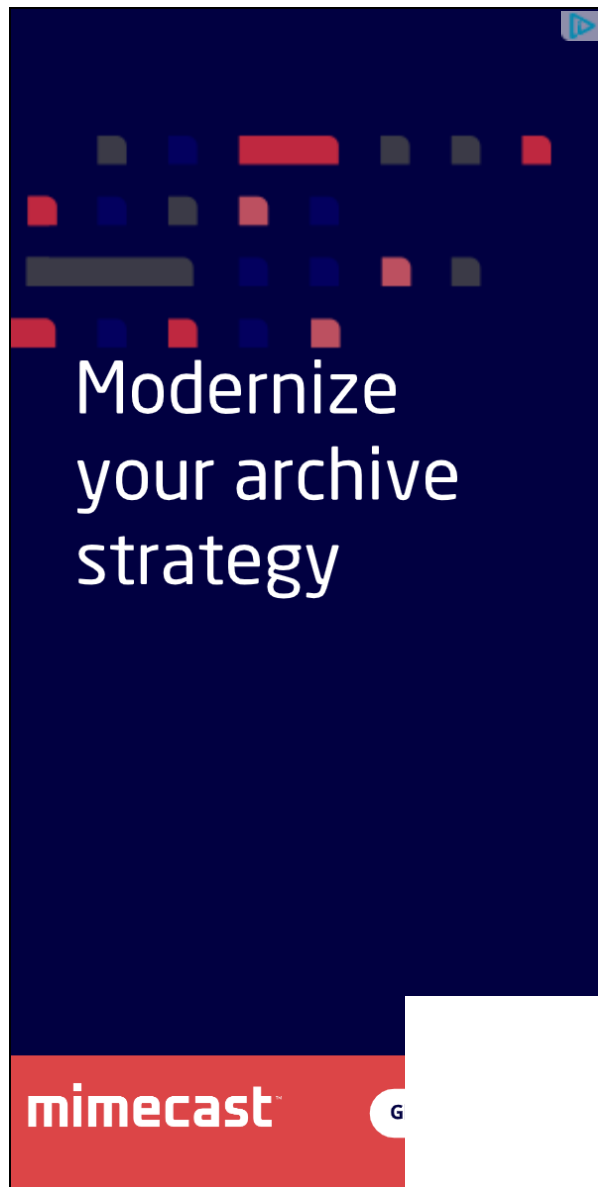
[Reply](#)

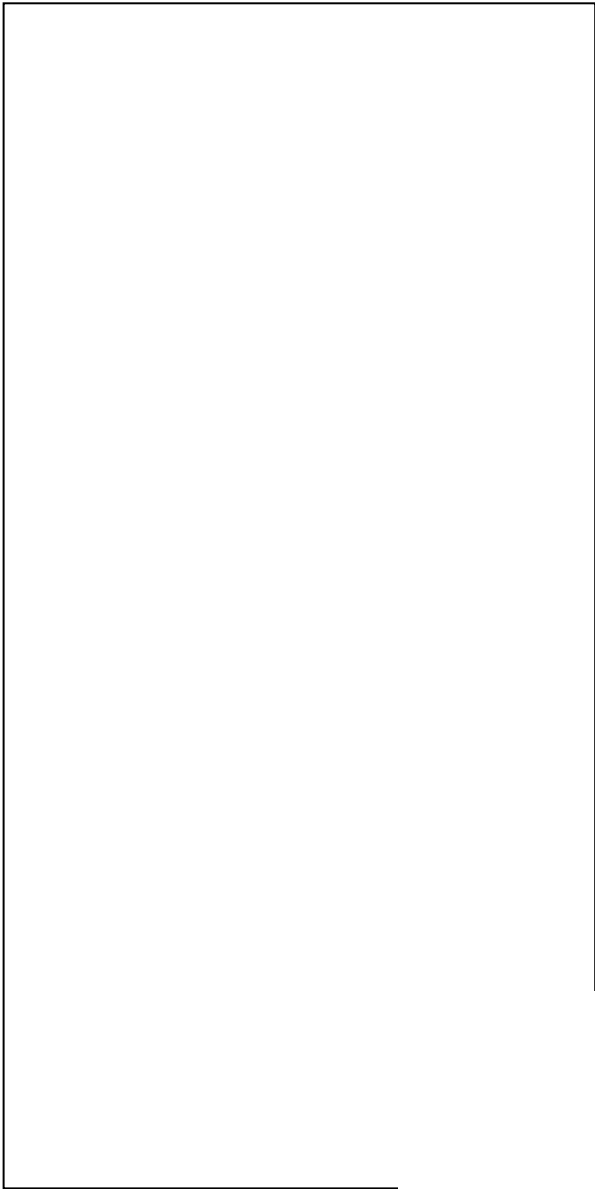
Leave a Comment

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment









HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- [About Me](#)
- [Contact Us](#)
- [Privacy policy](#)
- [Advertise](#)
- [Guest Posts](#)

Blogs

REST API Tutorial



Copyright © 2022 · Hosted on [Cloudways](#) · [Sitemap](#)