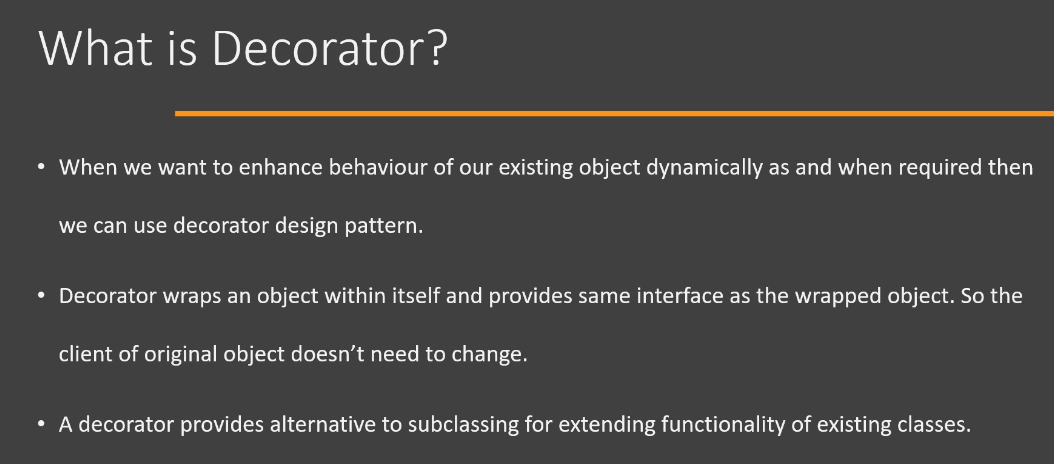
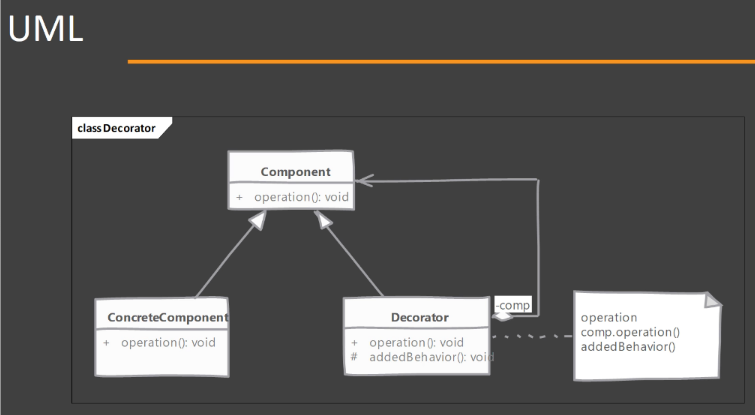
Decorator Design Pattern

1. Decorator – Introduction:

Decorator is a Structural design pattern that is very easy to understand, easy to implement, and it provides tremendous functionality for what it does. So let's jump into it and see what exactly a Decorator is. So whenever we want to enhance or add into existing object's behaviour and we want to do that dynamically, that means we want to decide at runtime whether we want this added functionality or not. Then we can use the Decorator design pattern and we're going to see how Decorator does that. So Decorator wraps the object whose functionality we want to enhance within itself, and then it provides the exact same interface(here, interface means the way it is exposed to the client, for example, its methods) which the object is providing. That means the code which is using our old object can use our Decorator. Now, since the Decorator has original object within itself, it can call the methods of that object. It can call methods on that object to provide the original functionality, plus it can add something new on top of that. All right, so this is how the Decorator design pattern works. Now, Decorator design pattern provides a good alternative to subclassing for extending the functionality of existing class. So you can decide which object needs this enhanced functionality, so you can pick and choose at runtime which objects can have this enhancement.



Now, let's look at the UML diagram of the Decorator design pattern. It's quite a simple UML diagram. First of all, we have a Component interface or abstract class, so this is the original interface or base abstract class that is used by our Client. If it(i.e., Component) is an abstract class or an interface, we would have a ConcreteComponent. This will be simply an implementation of our Component interface. And then we have Decorator. So Decorator itself extends or implements the Component interface, but it also contains an object of type Component. So whenever we call an operation() method on the Decorator object, first of all, it's going to use the object(i.e., of type Component) which it has within itself to provide the basic functionality**[**as you can see in the UML diagram below, comp.operation(), comp is an object of type Component. More specifically, here, comp is an object of ConcreteComponent. So, first operation() method of ConcreteComponent class will be called to provide the basic functionality**]** and then it can add on to that functionality as and when needed**[**as you can see in the UML diagram below, addedBehavior() is getting called which is a method of Decorator itself to enhance the functionality of object of ConcreteComponent class**]**. So this is the Decorator design pattern. It's quite simple and it's quite easy to implement.



A computer screen shot of a diagram

Description automatically generated

1. Decorator – Implementation Steps:

Now let's look at the steps that you would follow if you are implementing a Decorator. Now, it's quite a simple design pattern to implement, so there is nothing to it. We start by looking at our Component. So this could be a new interface that we write or this could be an interface that is already used by our Client. Then there is a ConcreteComponent class and this is a class whose object, Client is already using. Next, we define our Decorator. Now, we need to make sure that our Decorator implements the Component interface, which is used by client(i.e., Component interface is used by Client) and it's also going to need a reference to the object of ConcreteComponent class. So it's going to use that object of ConcreteComponent class in order to provide the basic functionality which the client is already using and it can then add on to that functionality as it needs. Next, Decorator itself can be abstract, so you can define your Decorator as an abstract class and then provide multiple subclasses or multiple child classes which act as decorators(i.e., Decorator's implementations). That way you have option to specialize each decorator(i.e., Decorator's implementation) for a different type of enhancement. So, this is all that is needed in order to implement Decorator design pattern.

A screenshot of a black and white text

Description automatically generated

1. Decorator – Hands-on Example UML:

Let's look at the example that we're going to implement in Java code. So this is a very simple design pattern to implement. What we're going to do is we're going to have a class called as Message(even interface or abstract class generates .class file upon compilation). So this(i.e., Message) will be an abstract class or an interface that has an abstract method called as getContent(). So it's going to give the message content as a String. There is one concrete implementation of this Component(i.e., Message) called as TextMessage. We have the implementation of getContent() method inside TextMessage class and it has a state 'msg'(class variable/field) which will be a simple string('msg' is a String type of variable/field). Now we're going to jump into implementing Decorators. So here in this UML diagram, I have given two different Decorators. First of all, we have Base64EncodedMessage. So what this Decorator is going to do, first of all, you will see that it is extending or implementing this Message abstract class or interface and it also has composition of TextMessage class object. So it's going to need an object of TextMessage class for it to work. This(i.e., Base64EncodedMessage) Decorator is going to encode the message content, so whenever we call getContent() method on the object of Base64EncodedMessage class, we're going to get the message content as base64 encoded String. On the similar lines, we are going to implement one additional Decorator called as HTMLEncodedMessage and what this is going to do, as you can guess, the message string will be HTML encoded and then returned to the Client. So this is a great example that you can see the benefit of using Decorator design pattern. You can do a lot of different things and you can keep on adding functionality without needing any change to the existing code(i.e., code of TextMessage class), by actually using these Decorators classes and you can decide whether to perform the additional functionality or not at runtime, because Base64EncodedMessage itself again is a Message. So any code that expects a Message can use the object of Base64EncodedMessage. So let's see how we can implement that in Java.

A screenshot of a computer

Description automatically generated

1. Decorator – Implementation:

Now we're going to see how we can implement Decorator in Java. So again, we have some simple classes that I have already implemented and these are exactly same as the UML that we saw previously. So, First, we have a Message interface. This is our Component interface which provides a single method called as getContent(). Then we have TextMessage. So this is our ConcreteComponent which implements Message and it simply holds a simple text message('msg' instance variable) within itself and it provides implementation of getContent() method by returning that single string('msg' instance variable). So if we jump back to the Client, so this is where we are using our Message object, you will see that we are creating a new object of TextMessage, providing it with some string and then we are calling its getContent() method and printing it. So if we run this code as it is, remember we have not yet introduced Decorator(Extract decorator\_begin.zip under Design-patterns-in-Java/Structural\_design\_pattern/3\_Decorator\_Design\_Pattern). This is simple plain Java. So we have a Message interface which is implemented by TextMessage and it provides some method and we get the output(i.e., The <FORCE> is strong with this one!) from this object(i.e., reference variable 'm' in the Client code). All right, now it's time to go and implement a Decorator. So here I have created an empty class called as HTMLEncodedMessage. So this will be our Decorator that encodes our string, that is stored in msg instance variable of TextMessage class, to HTML-encoded. So, to begin with, we are going to implement the Message interface which is used by our ConcreteComponent(i.e., TextMessage. TextMessage also implements Message interface) and implement/override the getContent() method. Now, we have seen that Decorator also needs an object of the actual ConcreteComponent. So what we are going to do is we are going to declare a private field(i.e., 'msg' instance variable) here(i.e., inside HTMLEncodedMessage class) of type Message and this 'private Message msg;' will be the object of ConcreteComponent that we need('msg' instance variable will hold the object of ConcreteComponent). Of course, we're going to need a constructor, inside HTMLEncodedMessage class, that accepts this ConcreteComponent class's object because without this object, we cannot work. Our Decorator cannot work without presence of this object because we are extending the functionality of ConcreteComponent. So we're going to define our only constructor that accepts an object of the class which implements Message interface(never provide the default constructor as well in the Decorator), and we're going to simply store it within ourselves(i.e., HTMLEncodedMessage). Notice, I have not used TextMessage as the type of the parameter inside the constructor of HTMLEncodedMessage class because we want to allow our Decorators to be recursively used. They should be able to be used recursive composition. That means we can pass another Decorator in the constructor of HTMLEncodedMessage class and our code will still work. All right, now let's implement the getContent() method inside HTMLEncodedMessage class. First of all, we can use the functionality that is provided by our original object(i.e., TextMessage class object). So we're going to call the getContent() method here on the Message type of object(i.e., 'msg' reference variable). But now it's time to add on to that functionality. So to do that, I'm going to use an Apache Commons library and use a class called as StringEscapeUtil. So this class is part of our Apache Commons library and we are going to use that class in order to encode or escape HTML. Call the escapeHtml4() method of StringEscapeUtil class and pass the String returned by getContent() method of TextMessage class. So this is how Decorator works. First of all, it performs the normal functionality on the object of ConcreteComponent class and get its behavior. In this case, we are getting the string that is the message or content of our Message object. And then we are going to add on to that returned message. In this case, we're going to escape that returned string so that it is HTML encoded. So now let's jump to our Client. So how we are going to use this(i.e., HTMLEncodedMessage) Decorator, we are going to create another Message reference variable named 'decorator' which is holding an object of HTMLEncodedMessage class and we are going to pass our existing object which is 'm' to this Decorator. Now we are going to use this Decorator as if it was a regular object because it implements the exact same Message interface. So I'm going to say 'decorator.getContent()'. Now let's run this code as Java application and you will see that we have the original message that is printed from line number 7 of Client code, however, from line number 10, we are printing the message which we are getting as an HTML encoded string. So you will see that(in the output of the run) these less than, greater than signs in the original message are replaced by a character that is compatible with HTML. So this is how we can use Decorators. I have another Decorator, which I have already implemented called as Base64EncodedMessage and again, it works on the similar lines. We are going to implement the Message and we are going to get the content from the object('msg') which is contained within it(i.e., Base64EncodedMessage). And then we are going to do base64 encoding. So to use this new Decorator(i.e., Base64EncodedMessage), what we can do here(i.e., inside the Client code) is I'm going to simply reuse our existing 'decorator' reference variable and then we are going to initialize our Base64EncodedMessage and then we are going to pass it our 'decorator' reference variable which is holding an instance of HTMLEncodedMessage class. So that is what recursive composition is. So we are giving an already decorated object to this Decorator(i.e., Base64EncodedMessage) and then we use this new ‘decorator’(i.e., reference variable) as if it was a regular object.

If you run this Client code now, Run as Java application, you will see that we get a Base64 encoded message, which was previously HTML encoded, which was originally a plain string. So this is how you can implement and use Decorator design pattern. This is quite powerful design pattern and once you start using it you will see that it has many things to offer. So that's all for implementation of Decorator design pattern.

1. Decorator – Implementation & Design Considerations:

Let's consider some of the implementation and design considerations that should keep in mind. Now, we have seen from the UML diagram that we have a requirement that our Decorator extends from Component abstract class. Now we have ConcreteComponent which is our real object and that is also extending from this common base class(i.e., Component). So if you have a large state defined in your Component base class, then that entire state is going to get inherited in Decorator objects and typically Decorators don't need that information because they add their own little thing to the behavior, they will use the class instance of ConcreteComponent to do something with the state of Component, so the state which they will be inherited from Component will be of no use. So try to avoid having a large state in your base class or base Component. Next, you should pay attention to equals() and hashCode() methods of Decorator. When using Decorator, you have a choice, and this is the decision that depends on the project that you're working, whether a decorated object is equal to the same instance without decorator. There is no right or wrong answer here. Only point that you should pay attention is that you are taking the equals() and hashCode() into consideration, especially if you are planning on using collections like Hashset and HashMap. You should decide what happens to the decorated object, whether it remains equal to the same object without decorator or not. Next, Decorators support what is called as recursive composition. What it means is that we can have a Decorator which contains another Decorator within it and this is possible because Decorators have implemented same interface(i.e., Component) as the original object(i.e., ConcreteComponent). So nobody can differentiate who is using this object, whether this is a Decorator or not. So they behave similarly. So each Decorator typically adds a small bit of functionality and you can end up with a nesting to a great level. So you will have a Decorator which again contains a Decorator within it and so on and so forth. And this type of code can become a little bit difficult to debug. So keep that point in mind when you are implementing Decorator.

A black and white text on a black background

Description automatically generated

Now let's look at design considerations. Now, Decorators are very flexible and powerful than inheritance because inheritance is static by definition. So you have a class which provides the functionality and if you create object of that class, you're always going to get that functionality. With Decorators, you can dynamically choose which objects can get the new functionality and which don't. So this is the benefit of using Decorator. So while you are designing your application, you can make use of this feature of Decorator. Next, Decorators should act as a kind of a skin over your object. That means they should provide small, meaningful functionality on top of your existing behavior. Do not try to change meaning of the operation/behavior in your Decorator. Okay, so if your original method and undecorated objects provides a certain type of behavior, decorators should not change the meaning of that behavior completely. So these are some of the points that you should keep in mind while designing applications using Decorator design pattern.

A screenshot of a computer

Description automatically generated

1. Decorator – Example:

Now let's look at an example of Decorator design pattern in Java class library. As you might have already known, the Java's I/O package is a really great example of this Decorator pattern. And here, we're going to look at BufferedOutputStream class, and this particular class can decorate any OutputStream type object and it adds the buffering capability to that particular OutputStream. And a benefit of this buffering is that it improves the disk i/o performance because it reduces the number of blocking i/o operations that are done. So let's look at the UML diagram of these particular classes. So first of all, we have the OutputStream and you can think of OutputStream as the abstract component in our Decorator. Then we have FileOutputStream, which is a concrete component, and then we have a class called as FilterOutputStream which is our decorator. The BufferedOutputStream is actually extending from our FilterOutputStream, and this BufferedOutputStream is also our decorator. So when we are working with BufferedOutputStream, we give that BufferedOutputStream's constructor an object of FileOutputStream or any child class of OutputStream. And this particular decorator(i.e., BufferedOutputStream) will add the buffering capability to that OutputStream. So there are many other classes in the Java's io package that you can look and you will find that they are great examples of Decorator design pattern where each decorate class adds a little bit of functionality on top of what is provided by the original object.

A screenshot of a computer

Description automatically generated

1. Decorator – Pitfalls:

Now let's look at pitfalls of Decorator design pattern. So using this design pattern often ends up with a large number of classes(i.e., Decorator classes) being added to your application, where each class adds a small bit of functionality. So at runtime, you often end up with lots of objects and we have one object nested inside another, and then we have another object which is nested inside that one, so on and so forth. And this type of arrangement can be quite difficult to debug, especially if you get any exception within one of the decorators. So that's one of the pitfalls and you should keep control on how many decorators you are implementing and what's a level of nesting you're using with your Decorators. Now, sometimes the newcomers or new learners of this design patterns start using it(i.e., Decorator design patter) as a replacement of inheritance in every possible scenario, because you can use composition and you always learn that composition is favoured over inheritance. But inheritance still has its place in object oriented software design. Always think of Decorators as a thin skin over your existing object, so you should not make large changes or significant changes to how the object behaves, how the original object behaves. We should not make that level of changes in Decorator. If you need those kind of changes where the behavior is significantly different, then you can use inheritance and override these methods, so, in this way, you will have a separate class and user will be aware that he's using a different class(i.e., object of different class). So these are some of the pitfalls of using Decorator design pattern in your application.

A screenshot of a computer program

Description automatically generated

1. Decorator – Summary:

Now let's look at the summary of Decorator design pattern. So we use this design pattern whenever we want to add a little bit of functionality on top of the existing functionality that is provided by the object. Now, Decorator has the exact same interface(i.e., it implements the same interface) as the object that it wraps, decorates or contains. So this is an important point and Decorator allows you to dynamically construct behavior at runtime using the composition, so you can decide which object is wrapped or which object is decorated. So that decision can be taken at runtime. Now a Decorator can wrap another Decorator. That means it supports recursive composition and the original Decorator(i.e., 1st Decorator) can have the actual object(i.e., concrete object) within itself. Now, Client of a Decorator object is unaware that it is using a decorated object, and that is possible because Decorator implements the exact same interface as the object that it decorates.

A screenshot of a computer

Description automatically generated