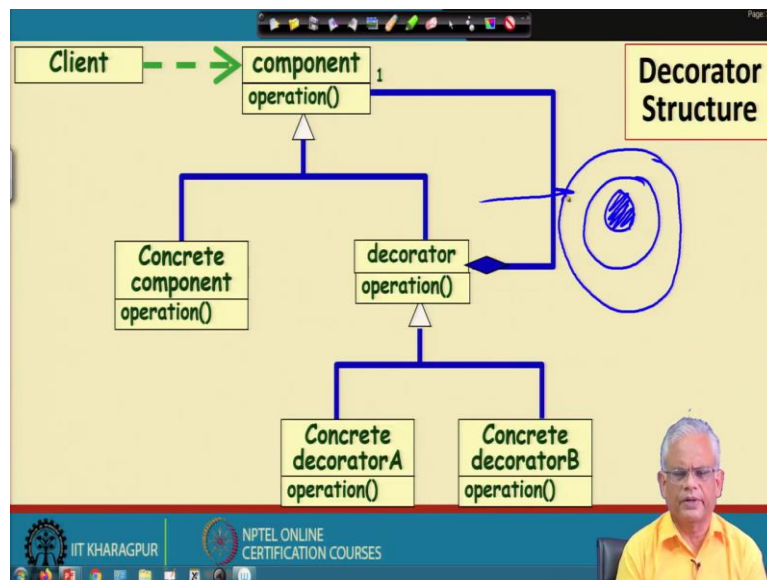


**Object Oriented System Development using UML, Java and Patterns**  
**Professor Rajib Mall**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 58**  
**Decorator Pattern II**

Welcome to this lecture.

In the last lecture we had started to discuss about the decorator pattern. We had said that the decorator pattern is a very powerful pattern. If we use it, we can add responsibilities to individual objects dynamically which is in contrast to static addition of methods or responsibilities to classes through inheritance. We had discussed about the basic structure of the decorator pattern and the class structure with the help of an example.

(Refer Slide Time: 01:07)



Now, let's recollect the class structure that we had discussed. If you remember the client interacts with the outermost decorator both the concrete component and the decorator they inherit from an abstract component or they implement an interface or they derived from the same abstract class component. And therefore, they have the same methods, for example operation is inherited by both the concrete component and the decorator.

The client cannot distinguish whether it is dealing with a concrete component or the decorator and therefore the decorators once added seamlessly sit on the concrete component and the client does not even know that it is dealing with something different. There can be different types of decorators but again they are derived from the same decorator class and therefore they have the same methods, of course they can add additional methods as well.

For the methods that are defined in the concrete component the decorator just forwards it to the concrete component and just see here that this is aggregation and there is one the cardinality is one (in the above slide), that means for a given component, we can add only one decorator at a time and each decorator adds additional functionality and the client interacts with the outermost decorator.

The class diagram is simple, we can understand it easily but it is a very powerful mechanism, let's try to look at some examples and see the advantage of the decorator pattern if we know this pattern well, we will have enough opportunity to apply this and make our application elegant and sophisticated.

(Refer Slide Time: 04:03)

**Decorator: Some Issues**

- **Consequences:**
  - + Responsibilities can be added/removed at run-time
  - + Avoids subclass explosion
  - + Recursive nesting allows multiple responsibilities
- **Implementation Issues:**
  - Interface conformance
  - Should use a lightweight class as a Decorator
  - Heavyweight classes make Strategy more attractive

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The decorator as we have been discussing we can add responsibilities at runtime that is dynamically we can add responsibilities and even we can remove responsibilities. Even though in the examples that we have discussed we have only added responsibilities but then it is also possible to remove the responsibility by just destroying one of the decorators.

Compared to adding responsibility through inheritance use of decorator avoid subclass explosion, we had seen through an example that if we can add responsibility to entire classes and for some specific applications, we might have hundreds of sub-classes it becomes extremely confusing for maintenance for development the programmer would have to look up.

And not only that these are permanently bound the responsibilities and therefore there will be several responsibilities of a class which the programmer does not need but still those methods

will be there just complicating the program. We can recursively nest multiple responsibilities as far as the implementation is concerned the decorator has the same interface as the concrete component. And each decorator should add only one or two small methods it should not be a heavy weight class; the decorator should not have drastically different and large responsibilities; if that is the case we need very different algorithms, very different responsibilities then the strategy pattern would be more meaningful rather than using the decorator.

(Refer Slide Time: 06:41)

The slide is titled "Decorator example: GUI". It contains the following text:

- Normal Java GUI components don't have scroll bars
- JScrollPane is a container with scroll bars to which you can add any component to make it scrollable

**// JScrollPane decorates GUI components**

```
JTextArea area = new JTextArea(20, 30);  
JScrollPane scrollPane =  
    new JScrollPane(area);  
contentPane.add(scrollPane);
```

The diagram illustrates the decorator pattern for a scrollable GUI component. It shows a "Scrollable client" containing a "View". This client is decorated with a "JScrollPane" which includes a "viewport", "Row header", "Column header", "Corner component", "viewport border", and "JScrollBar".

The slide also features the IIT KHARAGPUR logo and NPTEL ONLINE CERTIFICATION COURSES text at the bottom.

Let's look at some examples of uses of the decorator pattern. One of the areas we had remarked where decorator pattern is used profusely is in GUI development, graphical user interface development and also in the development of input output streams.

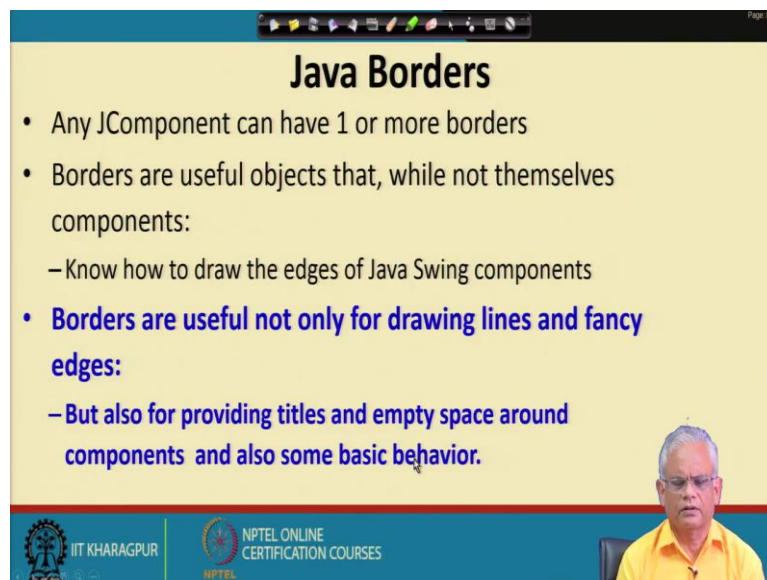
Let us first look at the GUI part. Typical GUI components don't have scroll bars, now in some situations we need scroll bars, for example we have lot of text and all are not visible we would like to have scroll bar so that the user can scroll up and down and look at the text. Java JScrollPane, or the widget is a container with scroll bars. And if we add any component to the JScrollPane, then that becomes scrollable and we will see that this is a direct application of the decorator pattern.

Now let's look at this example any GUI component like the JScrollPane can decorate it, let say we create a text area with 20 rows and 30 columns, so we create new JTextArea(20, 30) and area is the reference to that and then we create a scroll pane and we give area is the argument to it.

So, this is the constructor for the JScrollPane and it takes areas the argument and as a result the scroll pane will contain the area that is the text area which was created twenty columns and rows; 30 rows to which a scroll pane will get attached and then we can attach this to the content pane by the same mechanism.

So just see here (in the above slide) that there are two decorators which got added here, there is the scrollable client and text. There entire thing is the text but then our window cannot display everything this is the viewport only small part of the text is visible, now for the client to look at different parts of the text we need to give the scroll panes and that's what scroll pane do. The vertical and the horizontal scroll panes get attached here and now we can scroll and look at different parts of the text.

(Refer Slide Time: 10:30)



The slide is titled "Java Borders" and contains the following text:

- Any JComponent can have 1 or more borders
- Borders are useful objects that, while not themselves components:
  - Know how to draw the edges of Java Swing components
- **Borders are useful not only for drawing lines and fancy edges:**
  - **But also for providing titles and empty space around components and also some basic behavior.**

The slide also features a small video inset of a man in a yellow shirt in the bottom right corner, and logos for IIT Kharagpur and NPTEL Online Certification Courses at the bottom.

Another decorator is the java border. The scroll pane is a decorator and in GUI another popular decorator is a java border. Any JComponent we can attach border to that or decorate with border if we tell in more technical terms you can say that a component can be decorated with a border and we are implying that we are using the decorator pattern.

The borders themselves are not components but then they know how to draw the edges of the java swing components. The borders not only draw a line and fancy edges around the java swing component but also the borders provide title to a component and also some empty space around the component and also some basic behaviour.

The basic behaviour can be to minimize to maximize to fill the entire screen make it into an icon by minimizing to delete by pressing a cross and so on these are some of the basic behaviour that the border can give to a component.

(Refer Slide Time: 12:10)

**Decorator: Programming Example**

- We have a text object and ...
  - We want to add a border
  - At times we also want to add a scrollbar...

Border decorator  
Scrollbar decorator  
Original text object

For example, we may have a text object and we want to sometimes add a vertical scrollbar and. Sometimes we want to add a horizontal scrollbar.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now let's look at how to attach a border and a scroll bar (in the above slide). We want to attach two decorators. This is the original text, and then we want to attach a scroll bar and we want to attach a border (in the above slide). So, there will be two decorators we want to first decorate the original text with a scroll bar and then we will attach or decorate with the border. So finally, it would appear like this there is a scroll pane and then there is a border.

(Refer Slide Time: 13:02)

### Decorator Terminology

```
graph LR; Border --> Scrollbar; Scrollbar --> text;
```

- The objects refer each other like a linked list or chain of objects.
- The last in the list is the decorated object.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

If we see in terms of the class relations the border is associated with the scroll bar and the scroll bar is associated with the text or let say in the terms of object relations, we have the border which has a link to the scroll bar and the scroll bar has a link to the text and the terminology that we use the original text we call this as the decorated and the scroll bar and the border are the decorators and the client always interacts with the outermost decorator (in the above slide).

(Refer Slide Time: 14:00)

### Widget and Stream Examples

- Suppose you have designed a user interface toolkit and you wish to provide a choice of border or scrolling.
- **Widget** `aWidget = new BorderDecorator(new ScrollDecorator(new TextView()));`  
`aWidget.draw();`
- **Stream Example:**  
–cascade responsibilities to an output stream  
`Stream aStream = new CompressingStream(new ASCII7Stream(new FileStream("fileName.dat")));`  
`aStream.putString("Hello world");`

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES



Page: 10/10

Widget and Stream Examples

- Suppose you have designed a user interface toolkit and you wish to provide a choice of border or scrolling.
- **Widget** `aWidget = new BorderDecorator(new ScrollDecorator(new TextView()));`  
`aWidget.draw();`
- **Stream Example:** Draw class diagram...  
–cascade responsibilities to an output stream

```
Stream aStream = new CompressingStream(  
    new ASCII7Stream(new FileStream( "fileName.dat" ));  
aStream.putString( "Hello world" );
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let see the java code (in the above slide) we create a TextView and then we attach a scroll decorator that is a scroll bar to the text view and then we attach the border. So, we have created the TextView and given that as argument to the scroll decorator and then the entire thing is an argument to the border decorator.

You might have done such programming in java swing and if you know the decorator pattern then you will understand why you are doing like this and also the names given in the java swing components ‘border decorator’, ‘scroll decorator’ which hints that these are the direct use of the decorator pattern in the java swing-based user interface development.

And then if we say `aWidget.draw()` -- the draw method of the outermost component that is border decorator will be invoked and that in turn will pass the request to the scroll decorator and the scroll decorator will pass that into the text view decorator and those aspects which they can handle themselves they will not pass it. For example, scroll related `aWidget.scroll()` or something like that, that will be handled by the scroll decorator not be forwarded to text view but otherwise the border decorator will forward the request to the scroll decorator which will in turn forward the request to the text view.

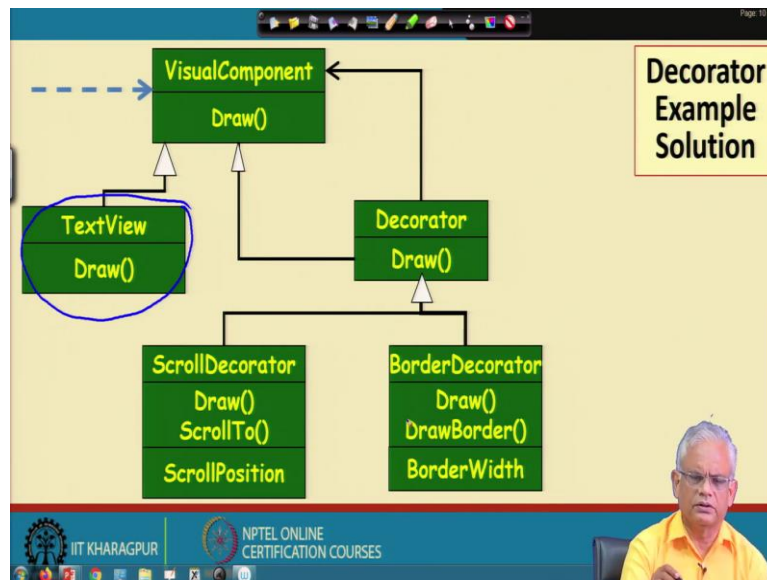
Now let’s look at an example of the java I/O stream. You might have done some programming using java I/O, the java input output and if you look at this code now, it will appear familiar and then you will realize that the code that you have been writing are actually the decorator pattern that has been used in the java I/O stream.

Just look at here we have created a file stream (in the above slide) a basic raw file stream here by giving the name of the file (`new FileStream(“filename.dat”)`) and then the ASCII 7 stream

(new ASCII7Stream()) interprets the raw file stream bytes as ASCII characters and then the compressing stream is a decorator which compresses it. So, if you put string hello world it will be converted to a file stream but first it will be compressed the ASCII 7 stream and then the file written to the file.

Above one is a clear example of a decorator pattern and after having done this how will class diagram look like? If we have defined the classes ASCII stream, compressing stream and the file stream how are these classes interconnected or structured if we know the decorator pattern, can we draw the class diagram? please try to draw the class diagram and we will just display that in the next slide.

(Refer Slide Time: 19:07)



Here text view is the basic component on which we can add various types of decorators both the text view and the decorator are derived from the visual component and there are various types of decorator one is the scroll decorator and the other is the border decorator.

The scroll decorator and border decorator also have the same interface as the visual component draw but they add additional capabilities. For example, scroll to add draw border. For those methods which they can complete the processing requirement, they just do it do not pass it to the internal object they are holding internally but for other ones like draw() they would pass it to the object that is inside.

So, you can have a text view to which you can either add a scroll bar and then a border or we can add a border and in a scroll bar and so on that will be decided in the run time and also, we

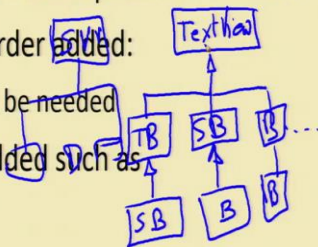


can add the border decorator twice. For example, on the text view we can add the border and then we can add the scroll bar and finally we can add another border these are all possible.

(Refer Slide Time: 20:56)

### Disadvantages of Inheritance

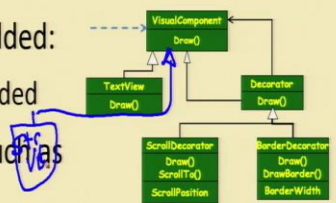
- Use of inheritance leads to an explosion of classes
- With another type of border added:
  - Many more classes would be needed
  - If another view were added such as StreamedVideoView:
    - It would double the number of Borders/Scrollbar classes
- **Use the Decorator Pattern instead!**



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

### Disadvantages of Inheritance

- Use of inheritance leads to an explosion of classes
- With another type of border added:
  - Many more classes would be needed
  - If another view were added such as StreamedVideoView:
    - It would double the number of Borders/Scrollbar classes
- **Use the Decorator Pattern instead!**



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So far in our discussion on the decorator pattern we have said that the decorator pattern gives a convenient option to add responsibilities to object which is advantageous in many situations in contrast to adding capabilities to an entire class through inheritance. Many times, the programmers get prompted or attracted to use inheritance because inheritance is simple to use appealing and for everything, they use inheritance.

But then if we use inheritance to add responsibilities that would lead to an explosion of classes, for example in this border example we want to the basic text view and then we derive

a class text view with border and then text view with border and scroll bar text view with just scroll bar and so on then we need many more classes just to draw that.

Let say we have text view class and we need another component if we are using inheritance then we will have text view with border. Now let us say we want text view with border and scroll bar and we want only text view with scroll bar and we want text view with scroll bar and then border and so on. Text view with a two borders and so on there will be lot of classes and if we use the decorator pattern we can add responsibilities to objects whenever necessary and also if we have another basic component like streamed video view in addition to text view, we also want to decorate a streamed video view, then that would be twice the number of classes here: we will have the streamed video view and then all these classes will be there.

But then if we use a decorator pattern then we just need one extra class here. If we add a new basic object and we use a decorator pattern then there is just incremental increase of only one class. Please try to draw the diagram where there we have two basic components the text view and the streamed video view and then we can add the border or a scroll bar to either of this. Please try to draw this and we will show the solution below.

So, we have the text view and we can have one more basic class the streamed video view. So just add one more class here basic class and these are the decorators can be applied to the screen video view as well (shown in the above slide).

(Refer Slide Time: 25:42)

The slide is titled "Decorator: Some Disadvantages" and contains the following content:

- When tempted to add many decorators:
  - A package may become hard to understand...
  - Like Java I/O streams!!! 🤔
- Solutions become complex:
  - A factory class may help

The slide also features a video feed of a speaker in the bottom right corner and logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES at the bottom.

But as do the decorators have some disadvantages? if we use the decorators recklessly, we had too many decorators then the code becomes difficult to understand. Like java I/O streams we can add many decorators if we look at the java book and we will find out there in the java I/O streams 60 odd decorators.

Now just imagine that we add these 60 odd decorators to a basic stream in various ways, then the person trying to understand the code would be very hard pressed because these are attached dynamically. You cannot just look at the code and then decide because you can even attach this dynamically.

So, we need to look at the runtime behaviour which becomes very difficult, the solutions become complex and if we have too many decorators we might use a factory class which will help us to create several decorators which will be manageable. We use a factory class to add lot of decorators but as long as we add only a handful of decorators like three, four, five and so on we don't need factory class.

Use of the decorator pattern provides a sophistication flexibility and our application can become elegant.

We are almost at the end of this lecture we will stop here and continue from this point in the next lecture.

Thank you.