Object Oriented System Development using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 57
Decorator Pattern I

(Refer Slide Time: 00:31)

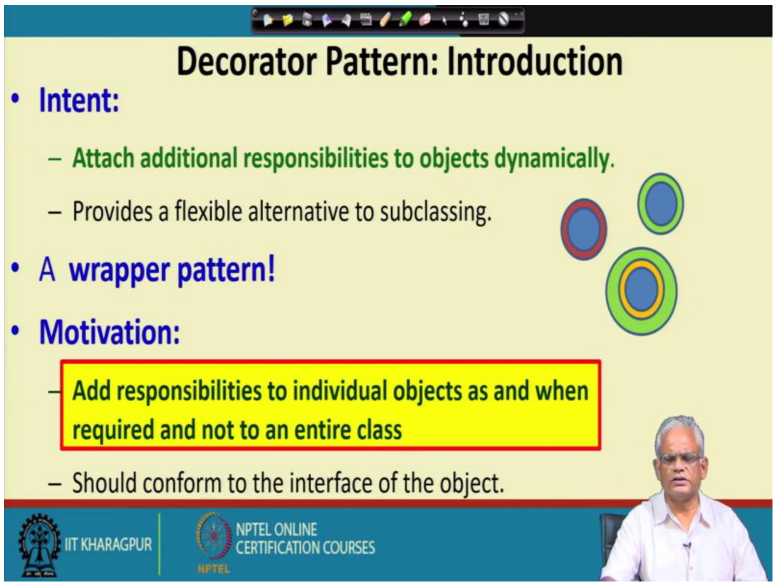
Decorator Pattern

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | 1

Welcome to this lecture.

In the last lecture, we looked at the proxy pattern, very useful pattern, it has large number of applications and now we will look at the Decorator Pattern. This is also a very important pattern, very powerful pattern. Many of the problems, if we can solve using the decorator pattern, then the solution becomes really elegant. Without the decorator pattern, it would be very inefficient and bad solution. Let's look at the decorator pattern.

(Refer Slide Time: 01:02)



Decorator Pattern: Introduction

- **Intent:**
  - Attach additional responsibilities to objects dynamically.
  - Provides a flexible alternative to subclassing.
- **A wrapper pattern!**
- **Motivation:**
  - Add responsibilities to individual objects as and when required and not to an entire class
  - Should conform to the interface of the object.

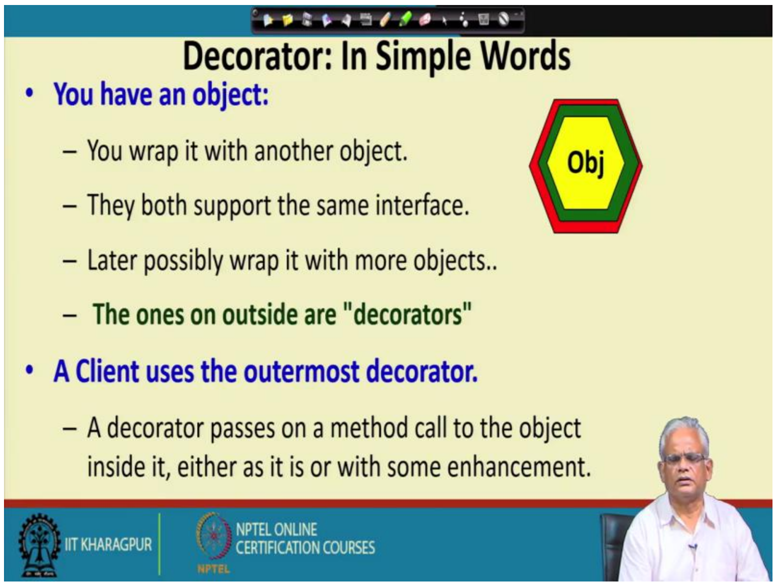
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The idea here is that we have many objects that have got created here. Now, during the runtime, if necessary, we want to add additional responsibilities to these objects. But how is that possible, because the responsibilities are basically methods and the class has only fixed number of methods. So, once you create the object, the methods are fixed, how do we attach additional methods to an object dynamically? So far, we have only been familiar with attaching methods by sub classing. But then, using sub classing, if we attach additional methods to a class, then they are bound to it permanently. But here, our intention is that we want to attach some methods to some of the objects and to other objects we want to attach other responsibilities.

Clearly, this is a wrapper pattern. We want to add responsibility to objects, specific objects that we want. We do not want to add responsibility to all the objects of a class that we know that by sub-classing, we attach additional responsibilities to all the objects of the subclass and here, once we attach the responsibility, it confirms to the same interface as the object.

Let say we want to attach some responsibility to the first object shown in the red that is additional responsibility (in the below slide). We want to attach a different responsibility to this object shown in yellow, another responsibility to this object which is shown in green (in the below slide). We might also like to add the green responsibility to this object after some time and so on. A very powerful concept let see how it is done.

(Refer Slide Time: 03:36)



Decorator: In Simple Words

- **You have an object:**
  - You wrap it with another object.
  - They both support the same interface.
  - Later possibly wrap it with more objects..
  - **The ones on outside are "decorators"**
- **A Client uses the outermost decorator.**
  - A decorator passes on a method call to the object inside it, either as it is or with some enhancement.

Obj
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The decorator in simple words is that you have an object and then you have another object, which wraps around this object. That means the wrapped object is the one which now becomes accessible to the user, the user cannot access this object anymore. Because another object will wrap around this and therefore it interacts with the one that is on the outside and that may pass on some method call to the object that is inside.

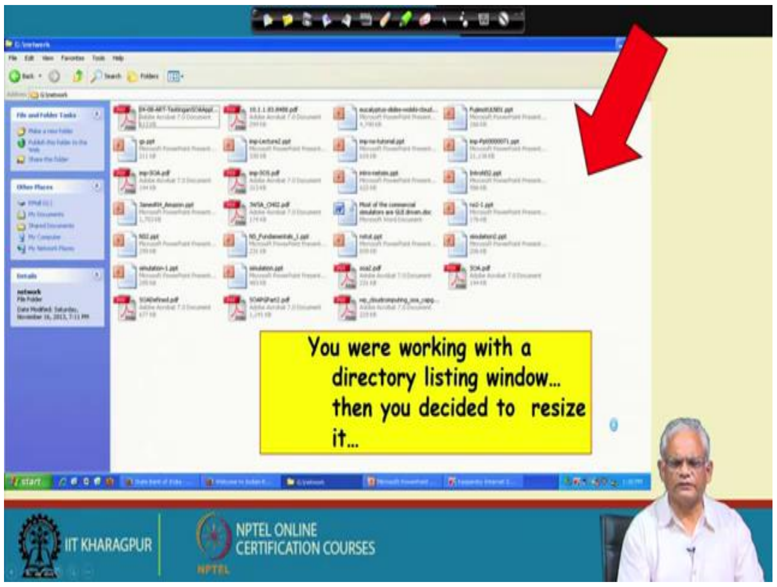
Obviously, the method that wraps around on the object will have the same interface as the object itself. So, this screen one has wrapped around the object. The clients can no more access the object, they have to use the green object because that is wrapping around the subject.

But then the client does not know really that the object has changed because they have the same interface. It invokes the same methods on the green and some of the new methods it answers by itself, it responds by itself. For the other old methods, the object it passes on to the object and therefore, the client of this class sees that some new capabilities of the object have materialized.

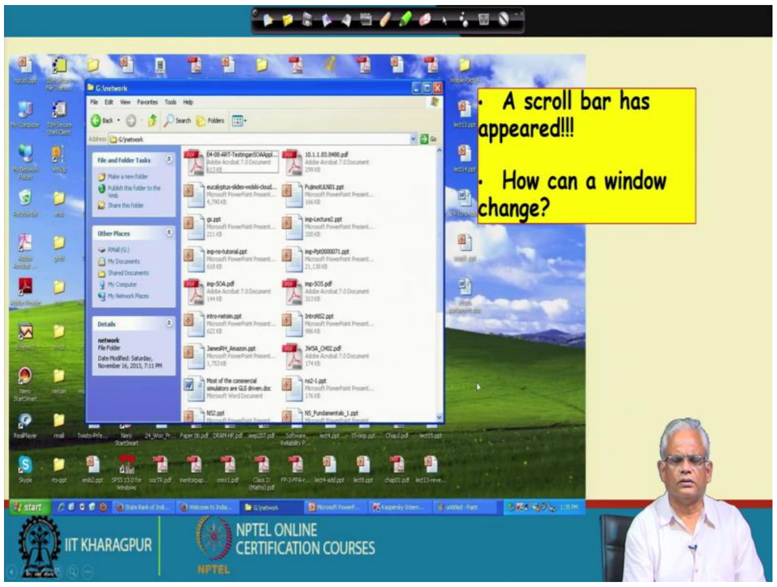
We can even have more objects wrapping around this object, red object has wrapped around here, the green is no more accessible directly, have to access the outermost object, the red. Thew red one whatever new methods it has it will answer or respond for those methods. For the other ones, it will delegate it to the green, the green in turn will answer some of that which has implemented and the rest it will pass it on to the yellow object.

The ones on the outside, this is the actual object and the green and red are the decorators. The client interacts with the outermost decorator and the outermost decorator first tries to handle those methods which it can and those methods which it does not implement, it passes on to the green and the green if it implements those methods, it will answer otherwise it will pass on to the real object.

(Refer Slide Time: 06:38)



You were working with a directory listing window... then you decided to resize it...

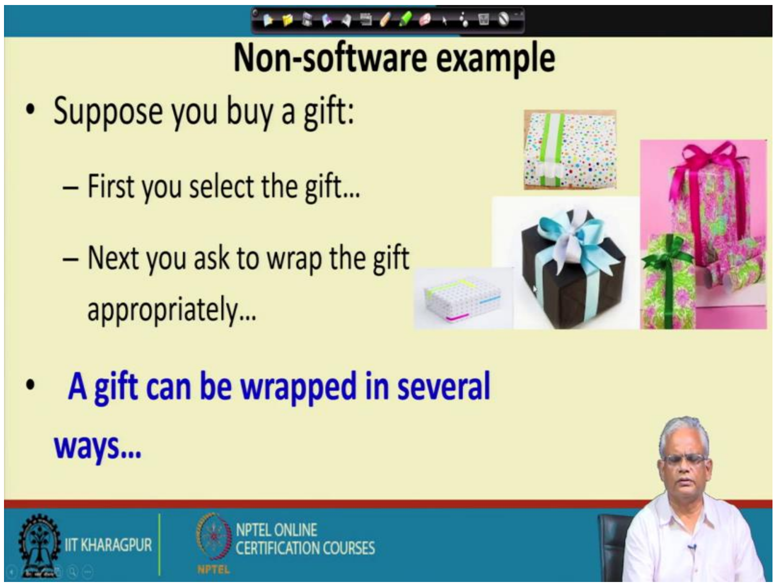


- A scroll bar has appeared!!!
- How can a window change?

Let’s look at one example. Let say you had an opened an window on your desktop, this is a window and then you wanted to resize. You are working on this window; you are seeing the listing and so on. Now, you want to resize it, you pulled the corner here down. But then by pulling the corner it has not only become smaller, but just see that there is a scroll bar which has got accessed because now all the files are not visible and therefore mysteriously a scroll capability has come to the same window. How is it that the basic window that existed now it has additional capability of scrolling, the full window if you see it did not have the scrolling capability, this is the full window (in the above slide).

All the files were visible and it did not have a scroll bar. But as you resized it, the files not visible and therefore a scroll bar has appeared. How come the same window object during runtime has acquired additional capability of scrolling. This is the use of the decorator pattern, we will see exactly what happens, how it gets the additional responsibility using the decorator pattern.

(Refer Slide Time: 08:38)



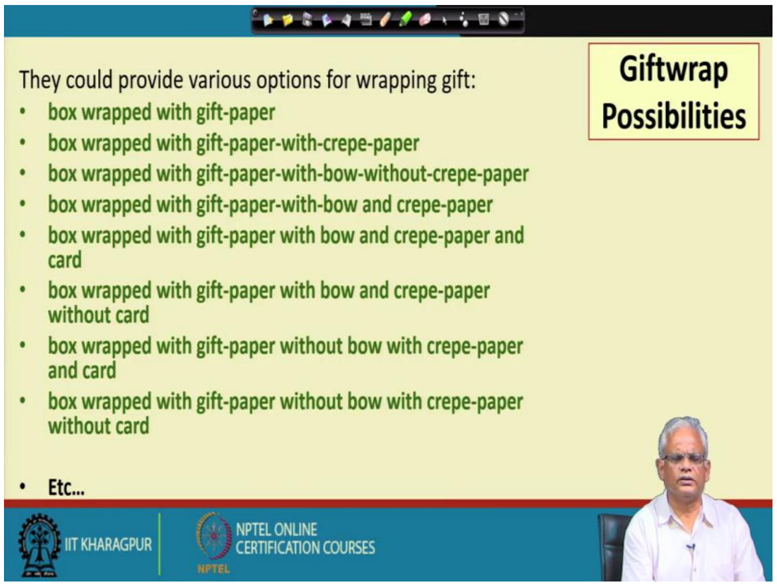
Non-software example

- Suppose you buy a gift:
  - First you select the gift...
  - Next you ask to wrap the gift appropriately...
- **A gift can be wrapped in several ways...**

Before we look at the nitty gritty of the decorator pattern, to better understand the pattern we will see an real life example. That is the motivation for the pattern, how it simplifies and helps and how without this pattern, the application will become very complicated. Let’s look at a non-software example which is easy to understand.

Let us say you went to buy a gift. You went to the gift shop and selected the gift and then you asked the shopkeeper to wrap the gift appropriately. But then you heard several options. It can be wrapped up in several ways maybe just a paper, maybe a paper with a tape, maybe with a bow and so on.

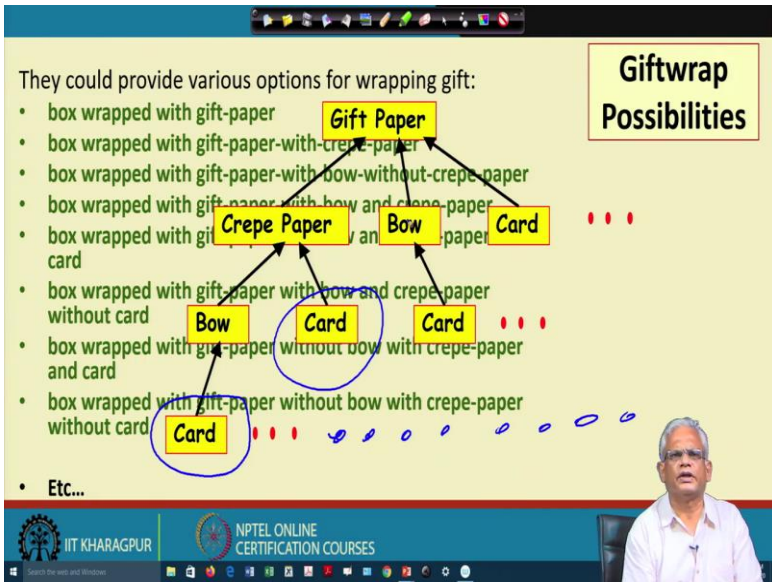
(Refer Slide Time: 09:52)



Giftwrap Possibilities

They could provide various options for wrapping gift:

- box wrapped with gift-paper
- box wrapped with gift-paper-with-crepe-paper
- box wrapped with gift-paper-with-bow-without-crepe-paper
- box wrapped with gift-paper-with-bow and crepe-paper
- box wrapped with gift-paper with bow and crepe-paper and card
- box wrapped with gift-paper with bow and crepe-paper without card
- box wrapped with gift-paper without bow with crepe-paper and card
- box wrapped with gift-paper without bow with crepe-paper without card
- Etc...



So, the different options, the gift wrap options that we will look at, maybe the gift boxes wrapped with only the gift paper. Maybe it is wrapped with the gift paper and crepe paper. Maybe it is wrapped with gift paper with bow but without crepe paper. Maybe it is wrapped with gift paper with bow and then the crepe paper. Maybe it is without card. Maybe it is with card and so on.

If we had all these as separate items in the shop that the shop you have these cupboards and in one cupboard, we have only gift paper. Another one where gift paper with crepe paper, another one cupboard with gift paper with crepe paper and bow, another one with gift paper, crepe paper and bow and card. Another one is only gift paper, crepe paper and card and so on.

You will see that the shop keeper will have to maintain at least two dozens of cupboard, no shopkeeper does that actually. It has one cupboard where it has the gift paper another one where it has the bow another one where it has a crepe paper, you can choose any one of them and then use it.

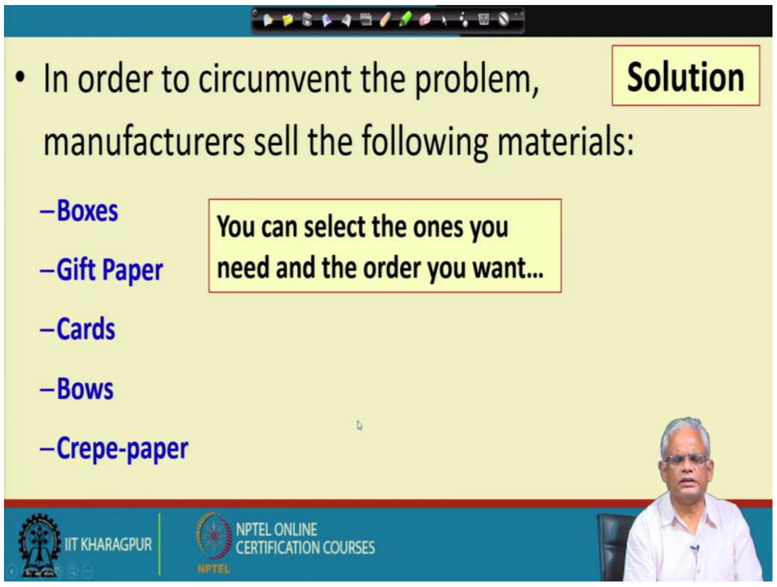
They are not statically fixed. You have not really attached statically the boat to the gift paper and then the card and so on that would have made the life of the shopkeeper and customers very difficult but we had the simple solution in the shop. If we did not have that simple solution, you would have different classes here, let say a gift paper with crepe paper bow and card this is one class and these are statically attached, but we did not do this actually, this is inheritance.

So basically, there will be many classes here and these are basically the cupboards and the shop. If you look at this cupboard, this has card, bow crepe paper and gift paper. So, this one is a gift paper with crepe paper and card. This is gift wrapper with bow and card. This is just gift paper and card and so on (in the above slide).

This is tactically attaching the different capabilities, the class hierarchy can become extremely large, if it was used in a shop, the number of cupboards will be extremely large, the customer will get confused, the shopkeeper will get confused.

The shopkeeper on the other hand keeps the gift paper, crepe paper bow and card in different cupboards. Only 4 cupboards and then as the customer says that what combination he needs and the order in which these need to be packed. He just selects them and puts them at runtime. It does not statically create these and then maintain it in the cupboard. The idea behind the decorator pattern is something very similar.

(Refer Slide Time: 14:37)



- In order to circumvent the problem, manufacturers sell the following materials:

Solution

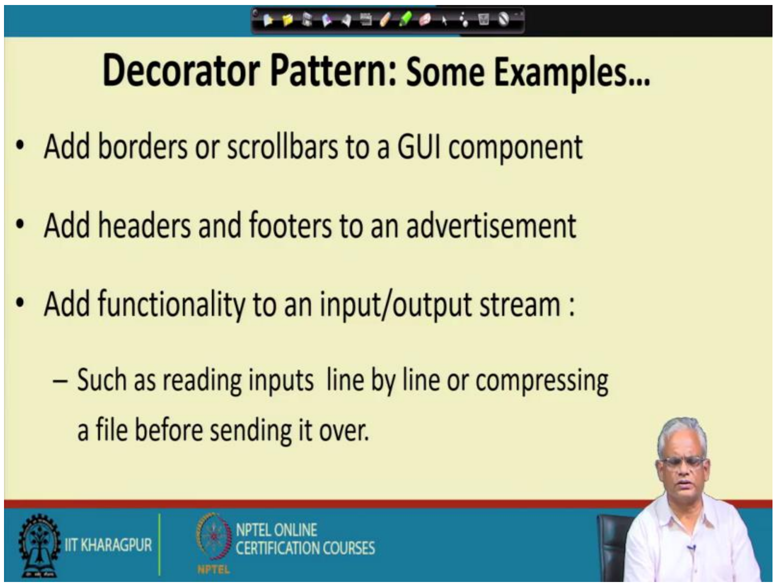
  - Boxes
  - Gift Paper
  - Cards
  - Bows
  - Crepe-paper

You can select the ones you need and the order you want...

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The shopkeeper just maintains the following cupboards: the boxes, the gift paper, card, bow and crepe paper as the customer chooses the ones that he needs and the order in which these are needed. The shopkeeper picks one from these and then puts them in proper order in runtime, they are not statically attached. Selects the ones that needed and the order in which it is needed.

(Refer Slide Time: 15:11)



Decorator Pattern: Some Examples...

- Add borders or scrollbars to a GUI component
- Add headers and footers to an advertisement
- Add functionality to an input/output stream :
  - Such as reading inputs line by line or compressing a file before sending it over.

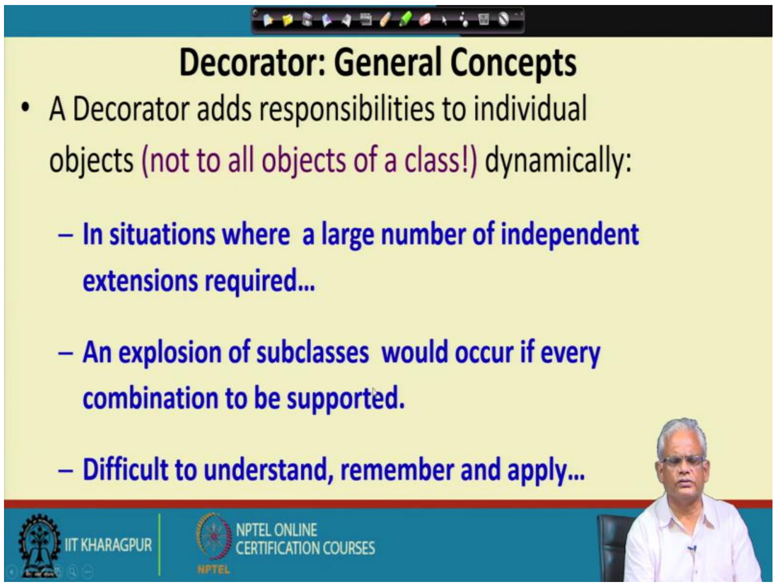
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The decorator pattern is very similar. Let us look at some examples. We had just seen that adding borders and scroll bars to GUI components, adding headers and footers to an advertisement. The advertisement object exists and then you might add in any number of headers to that in the runtime, any number of footers in the runtime.

It is not that there are different classes. One is advertisement with one header one footer, advertisement only header, advertisement without header without footer, advertisement with no header but two footers and so on. Here we can dynamically add as many headers and footers needed to an advertisement. Add functionality to an input output stream, the Java I/O we have been using.

The Java I/O profusely makes use of the decorator pattern. Here, you can add capability to the basic stream class. For example, compressing, we will just look at it that one of the largest uses of the decorator pattern. Decorator pattern is in the Java I/O. If we want to understand the Java I/O, we must understand the decorator pattern then only it will make sense to us.

(Refer Slide Time: 16:47)



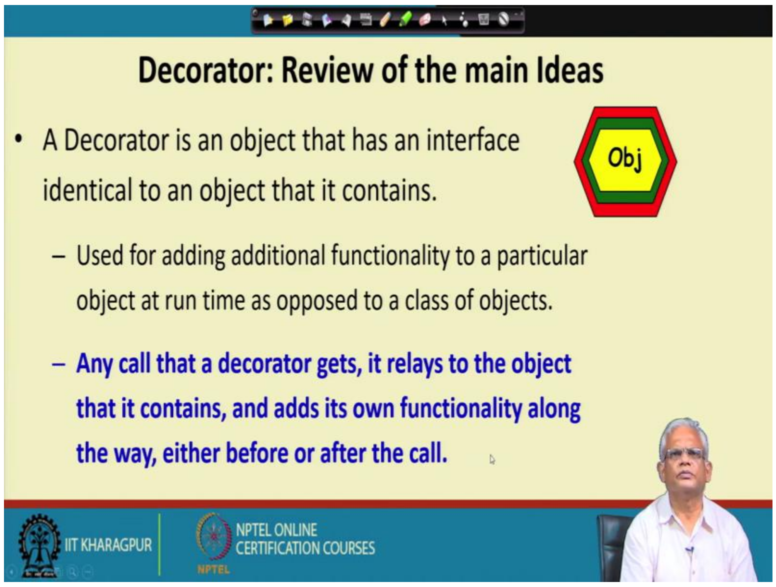
Decorator: General Concepts

- A Decorator adds responsibilities to individual objects (not to all objects of a class!) dynamically:
  - In situations where a large number of independent extensions required...
  - An explosion of subclasses would occur if every combination to be supported.
  - Difficult to understand, remember and apply...

Now let’s look at some general concepts. The decorator adds responsibility to objects. It is not sub classing; the sub classing will add responsibility to all objects of the class. But here we add responsibility to an object in runtime, dynamically as and when it is required. The decorator pattern is indispensable, when we need different capabilities to be added to different objects at runtime.

As we had already seen that if we really wanted to do this without the decorator pattern and by using sub classing, there will be explosion of subclasses easily. We will have thousands of sub-classes to deal with extremely confusing, difficult to remember and apply and this is a right candidate to use the decorator pattern will really simplify the application and make it elegant.

(Refer Slide Time: 18:05)



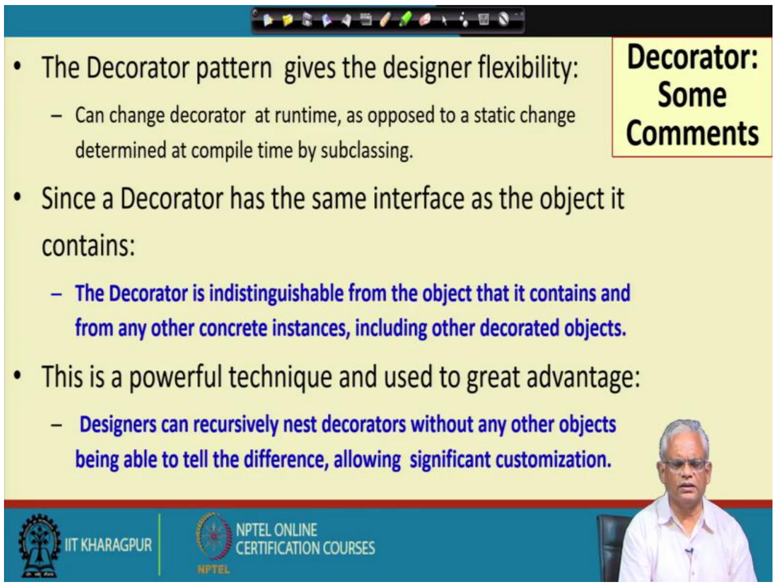
Decorator: Review of the main Ideas

- A Decorator is an object that has an interface identical to an object that it contains.
  - Used for adding additional functionality to a particular object at run time as opposed to a class of objects.
  - **Any call that a decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call.**

Now let’s review the main idea. A decorator is an object that has the same interface as the object it contents. A decorator adds some functionality to an existing object or an object with some decorators we will have an outermost decorator added which will have some extra capability and to that outermost, we can add another outermost decorator and so on.

We have already seen that we can keep on adding decorators and the client will interact with the outermost decorator which will try to handle the method invocations by itself if possible, otherwise it will pass on to the next decorator. As a decorator, the outermost decorator gets the call, it relays to the object that it contents and adds it to its own functionality either before or after the call.

(Refer Slide Time: 19:20)



Decorator: Some Comments

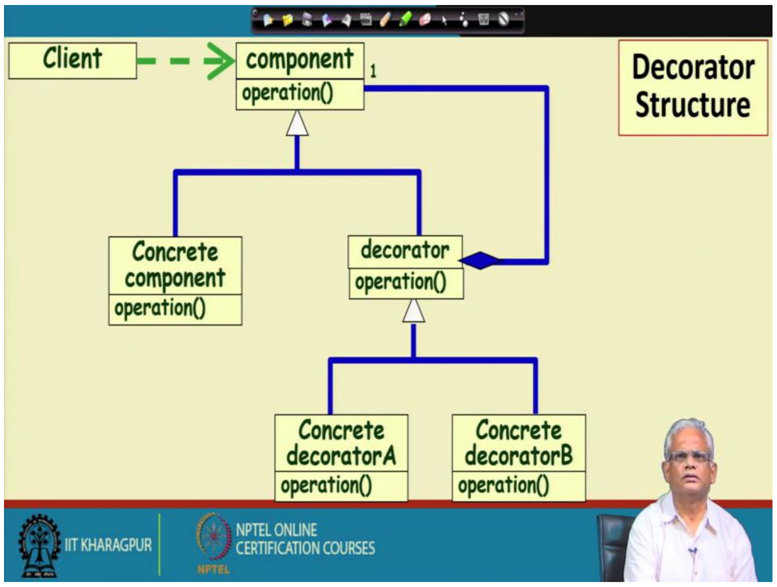
- The Decorator pattern gives the designer flexibility:
  - Can change decorator at runtime, as opposed to a static change determined at compile time by subclassing.
- Since a Decorator has the same interface as the object it contains:
  - The Decorator is indistinguishable from the object that it contains and from any other concrete instances, including other decorated objects.
- This is a powerful technique and used to great advantage:
  - Designers can recursively nest decorators without any other objects being able to tell the difference, allowing significant customization.

The decorator pattern, if we are aware, we get tremendous flexibility while designing an application. We can change the decorator at runtime: we can remove a decorator, add decorators, maybe add the same decorator twice, if necessary. We can add a different decorator, more decorators, remove some of the decorators as the run of the software proceeds.

The decorator gets added runtime as opposed to the static attachment of responsibilities that occurs by sub-classing. The decorator has the same interface as the object it contains. So, the client objects they do not even know that a decorator has come and taken place of the real object, just like we saw the window, mysteriously, additional capabilities appeared for the same object.

But it is not the same object, there is a decorator which has come and set on the real object. The decorator is indistinguishable from the real object which contains the concrete instances including the decorated objects which is a very powerful technique and can be used to great advantage. Designers can recursively nest decorators without any other object being aware of this or being able to tell the difference and allows significant customization

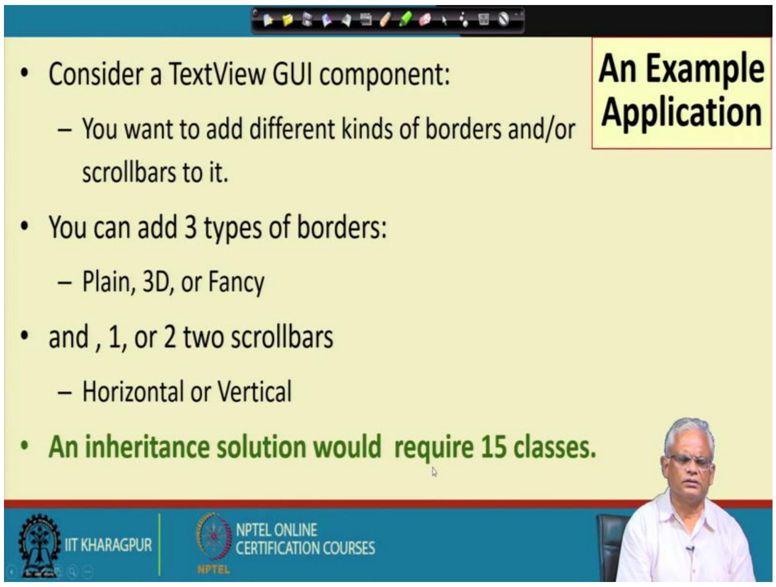
(Refer Slide Time: 21:19)



Now, let’s look at the class diagram. Component is an interface, the concrete component is the one which existed, the decorator also implements the component interface. Component can be abstract class or interface and there can be different types of decorators: concrete decorator A, Concrete decorator B, and so on and see here, that at any time, one decorator gets added.

On the decorated object, we can have other decorators get added one time at one. Requesting you to just compare this class diagram with the composite pattern class diagram. They appear very similar, if you look at them, but then there is one small difference. Functionally they are very different, their application is very different. I am just asking you to check the structural class diagram of the decorator pattern solution and the composite pattern solution and just try to identify that one difference between this diagram and the composite pattern diagram.

(Refer Slide Time: 23:10)



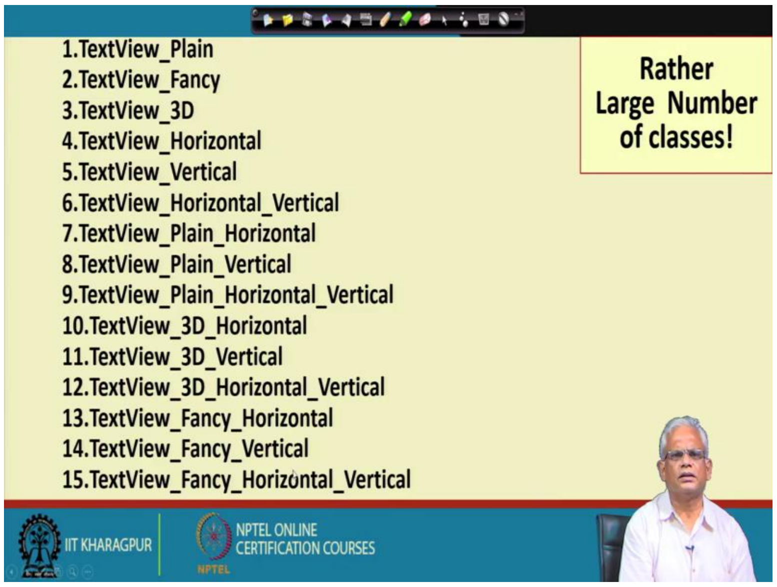
An Example Application

- Consider a TextView GUI component:
  - You want to add different kinds of borders and/or scrollbars to it.
- You can add 3 types of borders:
  - Plain, 3D, or Fancy
- and , 1, or 2 two scrollbars
  - Horizontal or Vertical
- **An inheritance solution would require 15 classes.**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now let’s look at an example application. Suppose you have a GUI component which has a TextView. It contains text and then you want to add different kinds of borders and scroll bars to it. Let say you can choose between three types of borders: plain, 3D and fancy and scrollbars, either one scroll bar or two scroll bar, the horizontal and vertical scrollbar and if you look at all possible combinations and provide this to solve this problem by using real classes that is a plain border, plain border with horizontal, plain border with both horizontal and vertical, plane border with only vertical, 3D border with horizontal, 3D border with vertical and so on. You will see that you require at least 15 classes. But when you can add the borders multiple times, let say two plain borders or a plain border and a 3D border and so on the number sub-classes that you will have to create is becomes very large. Nobody would like to do that and the application will become very difficult to manage, maintain, and understand. We will have to use the decorator pattern. But how do we use the decorator pattern? What will be the class diagram? Let’s look at the solution.

(Refer Slide Time: 25:16)

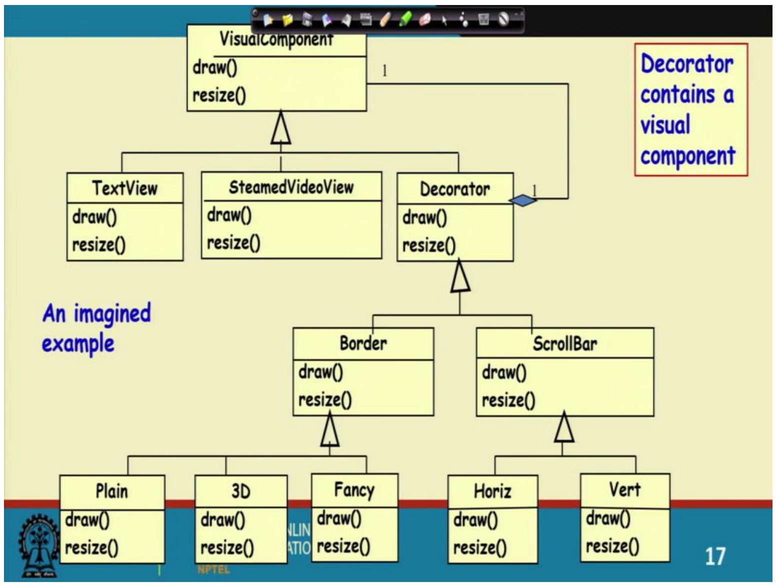


1.TextView_Plain
2.TextView_Fancy
3.TextView_3D
4.TextView_Horizontal
5.TextView_Vertical
6.TextView_Horizontal_Vertical
7.TextView_Plain_Horizontal
8.TextView_Plain_Vertical
9.TextView_Plain_Horizontal_Vertical
10.TextView_3D_Horizontal
11.TextView_3D_Vertical
12.TextView_3D_Horizontal_Vertical
13.TextView_Fancy_Horizontal
14.TextView_Fancy_Vertical
15.TextView_Fancy_Horizontal_Vertical

Rather Large Number of classes!

These are the 15 classes sub classes that you might have to use for above example. If you wanted to statically attach these responsibilities the TextView with just a plain border, TextView with a fancy border, TextView with a 3D border, TextView with only horizontal scroll bar, vertical scroll bar, TextView with horizontal and vertical scroll, TextView with a plain border with horizontal scroll bar and so on then required 15 classes. But if you wanted to allow repetition of some of the borders, maybe a plane with a 3D and also scroll bar and so on, the number will be much large and writing an application with that many classes to choose from the programmers' life will be really difficult. But what you can do is use the decorator pattern. As we have already seen that the decorator forwards request to the component and performs additional responsibility.

(Refer Slide Time: 26:43)



So, the solution will look something like this (in the above slide) that we have visual component and then we have decorator border scroll bar, plain, 3D, fancy, horizontal, vertical and here TextView and we also have a streamed video view. The other problem we didn't have. If we had a stream video view in the other problem, then the count 15 will have to be increased tremendously.

So, here both TextView, streamed video we can add all this border and scroll bar anything of them dynamically. It is very powerful pattern. We will look at more examples. So, that we become real familiar in applying this pattern to real problems.

We are almost at the end of this lecture. We will stop here and continue in the next lecture, where we will discuss about the decorator pattern with more examples and also the Java code. Thank you.