Object - Oriented System Development Using UML, JAVA and Patterns

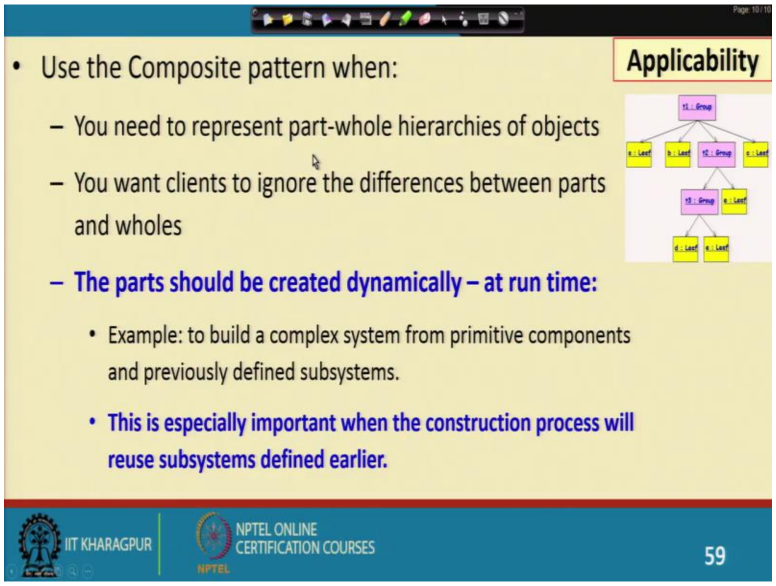
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 50

Composite Pattern - II

Welcome to this lecture! In the last lecture we were discussing about the Composite Pattern, very important pattern. If you know the pattern you can effortlessly come up with a good solution to the problem and if you are not aware of this pattern you would struggle, make many iterations and with lot of prompting maybe you can come to the right solution and therefore it is good to know this pattern.

(Refer Slide Time: 1:14)



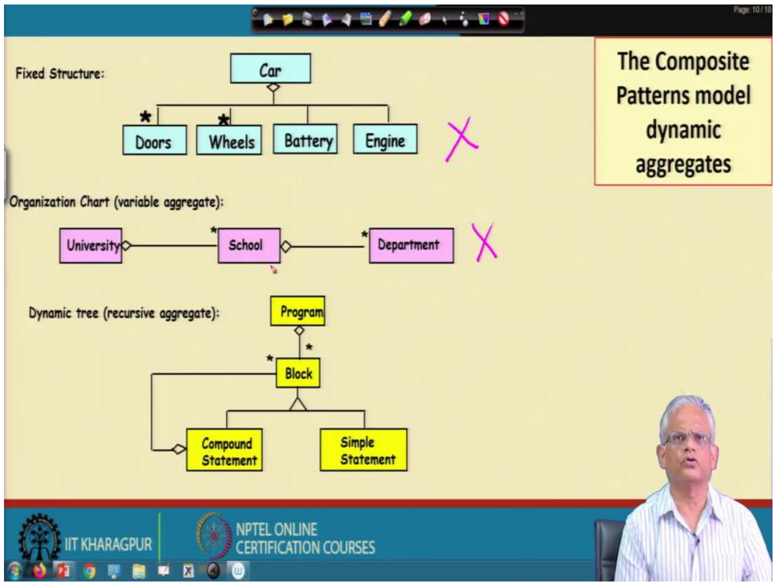
Applicability

- Use the Composite pattern when:
  - You need to represent part-whole hierarchies of objects
  - You want clients to ignore the differences between parts and wholes
  - The parts should be created dynamically – at run time:
    - Example: to build a complex system from primitive components and previously defined subsystems.
    - This is especially important when the construction process will reuse subsystems defined earlier.

At the end of the last lecture we were discussing about the applicability of this pattern and we commented that this pattern is applicable when there is a path whole hierarchy of objects. This is the first requirement that there appears to be a tree like hierarchy. But then that is not sufficient or enough to use the pattern just because there is a tree hierarchy.

The other requirement is that the parts are created dynamically and formed in two groups. That is, we can form groups from primitive elements and then form larger groups from primitive elements and existing groups and still larger groups and so on. Otherwise if the second condition is not there that we form larger groups at runtime then using this pattern is an overkill, let us look at an example.

(Refer Slide Time: 2:19)


Fixed Structure:
Car
Doors
Wheels
Battery
Engine
Organization Chart (variable aggregate):
University
School
Department
Dynamic tree (recursive aggregate):
Program
Block
Compound Statement
Simple Statement
The Composite Patterns model dynamic aggregates
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Let us say we have a pothole hierarchy here. A car consists of many doors, many wheels, battery and engine is a hierarchy but should we use the composite pattern here? No, here the groups are not formed dynamically, there is a fixed structure here, there is no dynamic forming of groups. So, applying the composite pattern here will be an overkill and make the design more complicated than required because the problem is very simple.

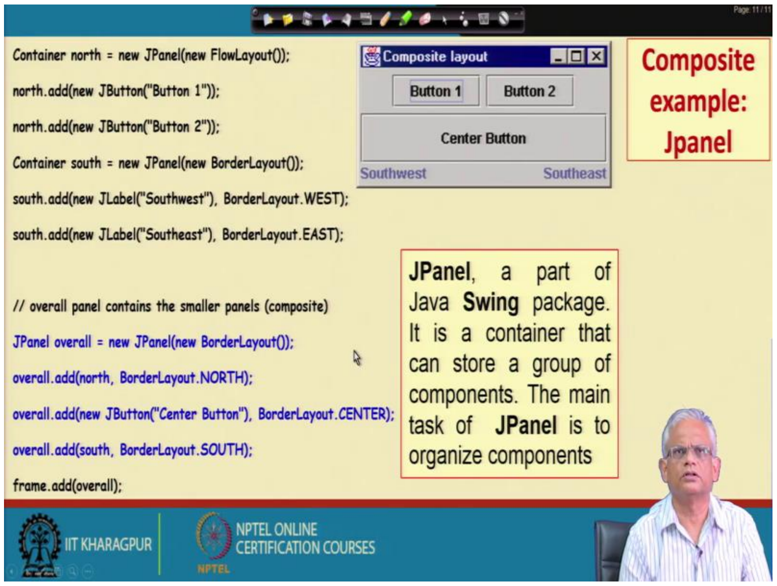
Similarly, let us see that there is hierarchy here, a university contains many schools and each school contains many departments. Now, what about this? This is also a hierarchy and school may contain many departments some schools and the university may contain many schools and so on.

But again, we will not apply the composite pattern here, because we do not form the groups here dynamically, this is a fixed structure here. If you see the problem here that we have a program written where we have some groups here called as blocks, they contain simple elements or other blocks and form larger blocks and so on.

The program contains some blocks which contain simple statements, we can form larger blocks by using the simple blocks and simple statements and still larger blocks using the other blocks and simple statements and so on. This is the situation where the composite pattern should be used.

And these two places if we try to use the composite pattern that would be an overkill, the simple class structure is presented here is enough, we do not need a composite pattern here, because we dynamically do not form larger groups and group containing more groups and so on.

(Refer Slide Time: 5:24)



Composite example: Jpanel

```
Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));
Container south = new JPanel(new BorderLayout());
south.add(new JLabel("Southwest"), BorderLayout.WEST);
south.add(new JLabel("Southeast"), BorderLayout.EAST);

// overall panel contains the smaller panels (composite)
JPanel overall = new JPanel(new BorderLayout());
overall.add(north, BorderLayout.NORTH);
overall.add(new JButton("Center Button"), BorderLayout.CENTER);
overall.add(south, BorderLayout.SOUTH);
frame.add(overall);
```

JPanel, a part of Java **Swing** package. It is a container that can store a group of components. The main task of **JPanel** is to organize components

Now, let us see the use of the composite pattern in Java GUI, that is Java swing, those who are not very familiar with the Java swing there is a Java user interface. The JPanel is a part of the java swing package, it is a container that can store a group of components and the component can be primitive or composite and the main task of the JPanel is to organize the component.

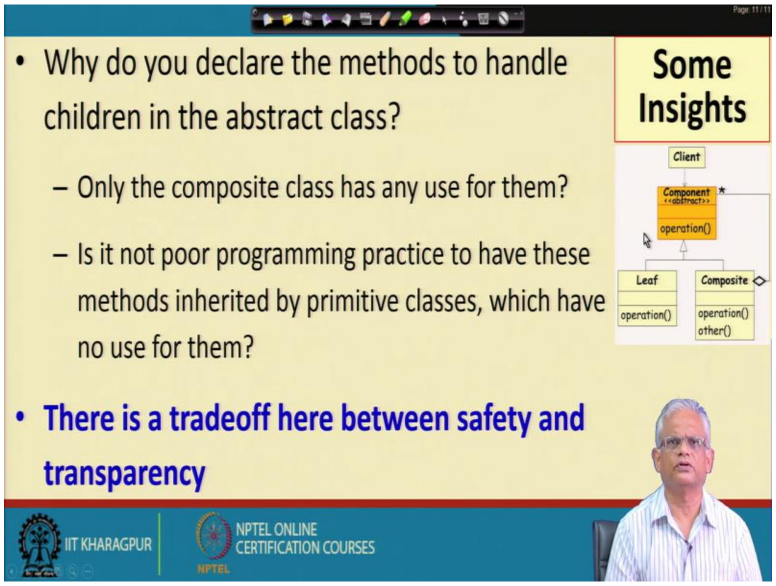
As the problem states, can see that, it is straight application of the composite pattern in the Java swing package, the composite layout here contains some primitive elements like two buttons and there is a center button and then we can create a more or a larger composite using this and so on, let us see the code here.

We have a container north which is a JPanel and we define the flow layout for that and then we add a J button here, button 1 and then to the north which is a container or a composite we add button 2 and then we create a container south and this is again a JPanel and then we are to the south label Southwest and then we are Southeast.

And then in the JPanel over all, overall JPanel which is new JPanel here overall, so to that we add this north which is a composite and also we add the primitive element, a center button, so

Southwest and Southeast these are composite and then the final one we add the north which contains, which is a composite of two buttons, we add the center button and also we add the south. And then to the frame we add the overall. So, we can see here that we are grouping here elements and adding to a composite forming larger composites and so on, this is an application of the composite pattern.

(Refer Slide Time: 8:24)



- Why do you declare the methods to handle children in the abstract class?
  - Only the composite class has any use for them?
  - Is it not poor programming practice to have these methods inherited by primitive classes, which have no use for them?
- **There is a tradeoff here between safety and transparency**

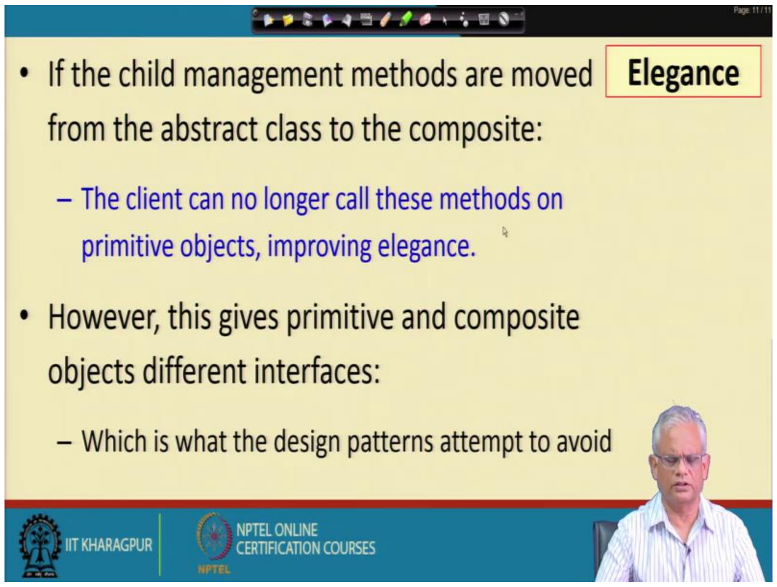
Some Insights

Now, how do you declare the methods to handle the children in the abstract class? Because if we add the children in the abstract class then these are only useful for the composite, just see here that the component is the abstract class and should we add the children management operations on the component?

Just remember that the children management operations are useful only by the composite, the leaf they do not have children and they should not implement these operations. Would it be a poor programming practice if we have the children management operations defined in the abstract class? What do you think? That leaf will not have any use for them and would it be a poor programming?

There is a tradeoff here, one is that whether the client handles all the elements in the design similarly or he has to do anything separate? So, there is a tradeoff here, that there is a violation of the design principle that we are trying to define some operations in the abstract class which the leaf has no use for them. And the other way we are trying to have a uniform operations defined for the client.

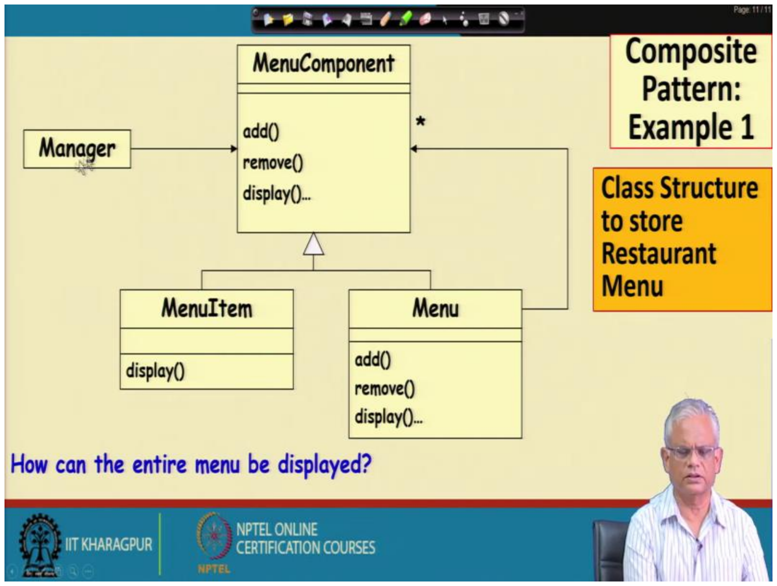
(Refer Slide Time: 10:22)



- If the child management methods are moved from the abstract class to the composite: **Elegance**
  - The client can no longer call these methods on primitive objects, improving elegance.
- However, this gives primitive and composite objects different interfaces:
  - Which is what the design patterns attempt to avoid

If the child management methods are moved from the abstract class to the composite so they will become different basically, then it looks like more elegant but then the primitive and the composite get different interfaces so you can see the tradeoff here, one is that we are trying to debate between the elegance of the design and also from the clients convenience whether it appears uniform or not.

(Refer Slide Time: 11:02)



Composite Pattern: Example 1

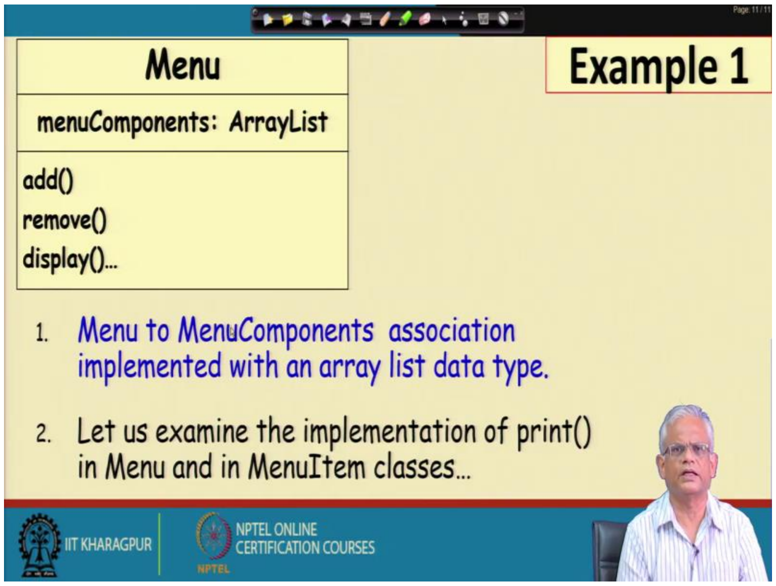
Class Structure to store Restaurant Menu

How can the entire menu be displayed?

We will go for client’s convenience, and we will sacrifice little bit of design elegance and we will have typically support the client management also in the abstract. So, now let us look at an example, let us say we have a restaurant menu. The menu is grouped, there are some primitive menus which are vegetarian, non-vegetarian, there are Chinese dishes, there are English dishes and so on, under that there are primitive elements.

So, the menu has different sections, now we want to have a solution, a design solution to this menu we can straight away use the composite pattern, the menu component is abstract class where you can add remove display menu, the menu item is a primitive one and the menu contains many menu components, the manager interacts with the menu components.

(Refer Slide Time: 12:25)



Example 1

Menu
menuComponents: ArrayList

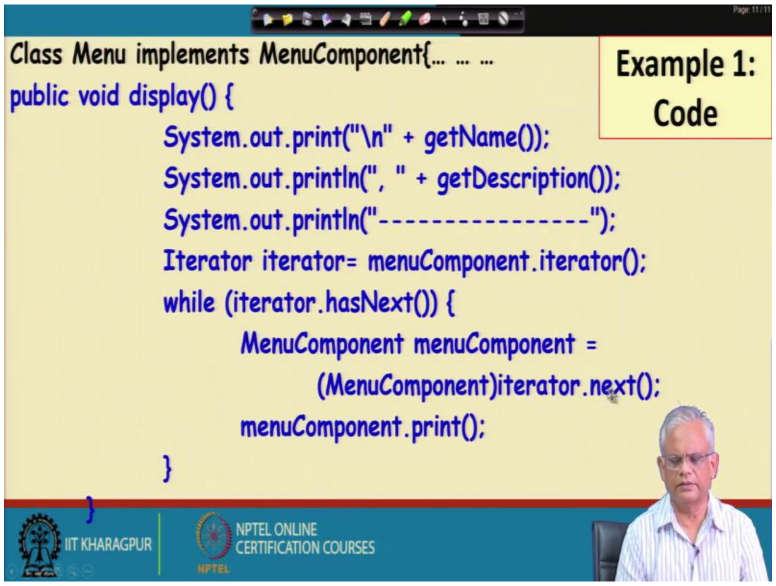
add()
remove()
display()...

1. Menu to MenuComponents association implemented with an array list data type.
2. Let us examine the implementation of print() in Menu and in MenuItem classes...

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, on the implementation side, for the composite menu how will it store the children? A good solution is to use a ArrayList and then we can add, remove, etc. very easily. Now, let us examine how to use the print function, let us say the manager wants to print the menu.

(Refer Slide Time: 13:00)



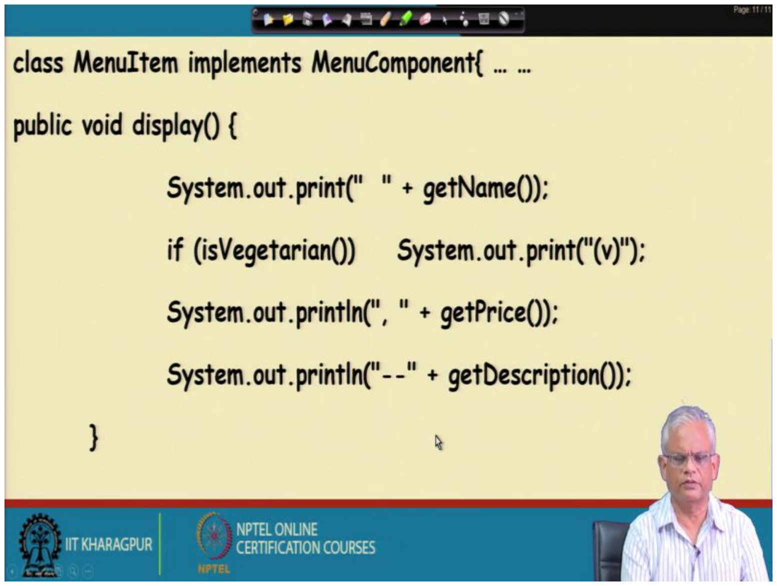
Example 1: Code

```
Class Menu implements MenuComponent{... ... ...
public void display() {
            System.out.print("\n" + getName());
            System.out.println(", " + getDescription());
            System.out.println("----------------");
            Iterator iterator= menuComponent.iterator();
            while (iterator.hasNext()) {
                  MenuComponent menuComponent =
                        (MenuComponent)iterator.next();
                  menuComponent.print();
            }
}
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, the class menu implements menu component, menu component is the interface and then the display, it prints, it uses an iterator and then for all the menu components it calls the print as long as there is a next element it just calls the print.

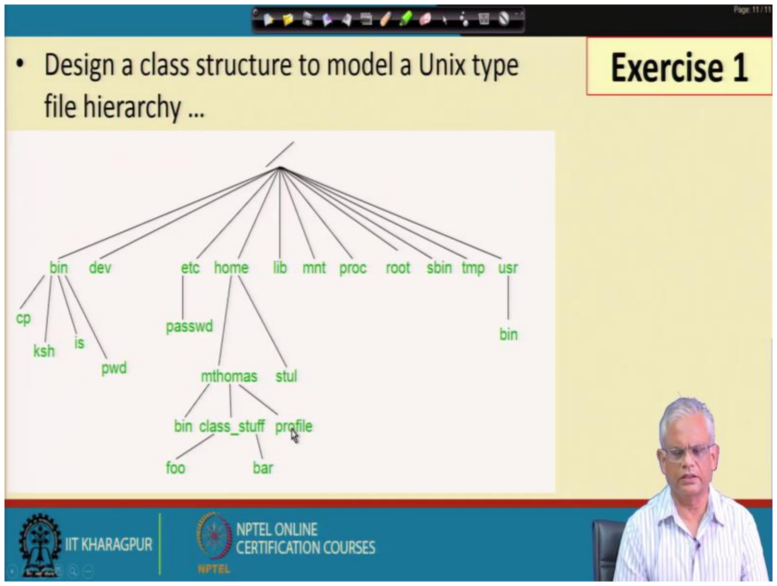
(Refer Slide Time: 13:35)



```
class MenuItem implements MenuComponent{ ... ...
public void display() {
            System.out.print("  " + getName());
            if (isVegetarian())    System.out.print("(v)");
            System.out.println(", " + getPrice());
            System.out.println("--" + getDescription());
    }
```

And for the primitive one, it just displays the price, name of the menu, price and some description. So, the primitive element just displays this and the composite menu invokes the print of the primitive elements.

(Refer Slide Time: 14:00)

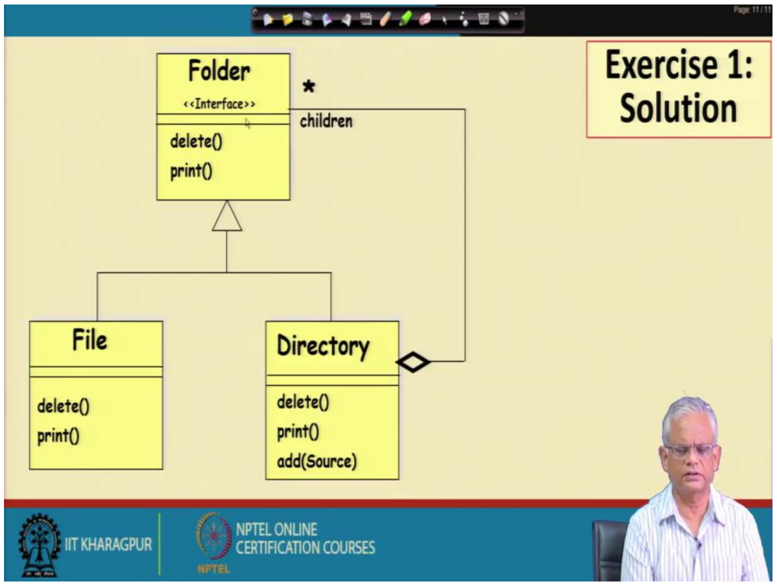


- Design a class structure to model a Unix type file hierarchy ...

Exercise 1

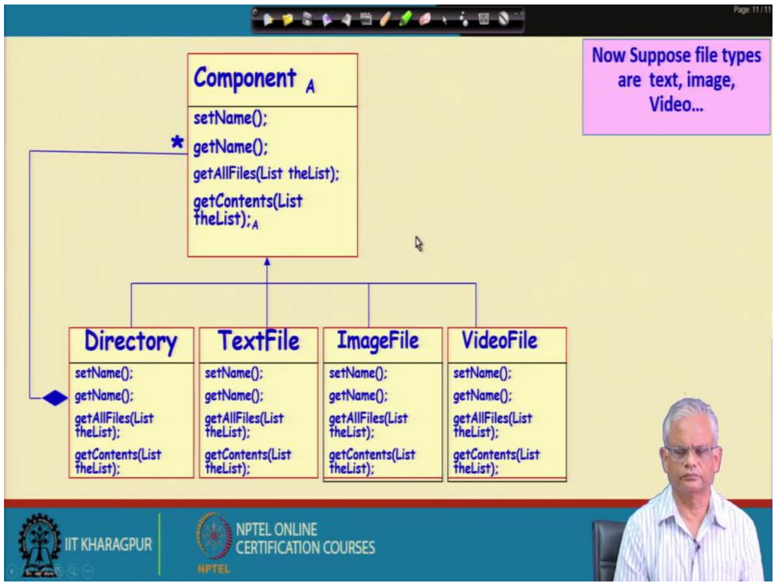
Now, let us do another exercise, let us say we have a file hierarchy, a Unix type file hierarchy, where a directory can contain files and other directories. Look at the root directory it contains some of the files and some directories. And the directories in turn can contain more directories and files. So, what will be the solution here?

(Refer Slide Time: 14:42)



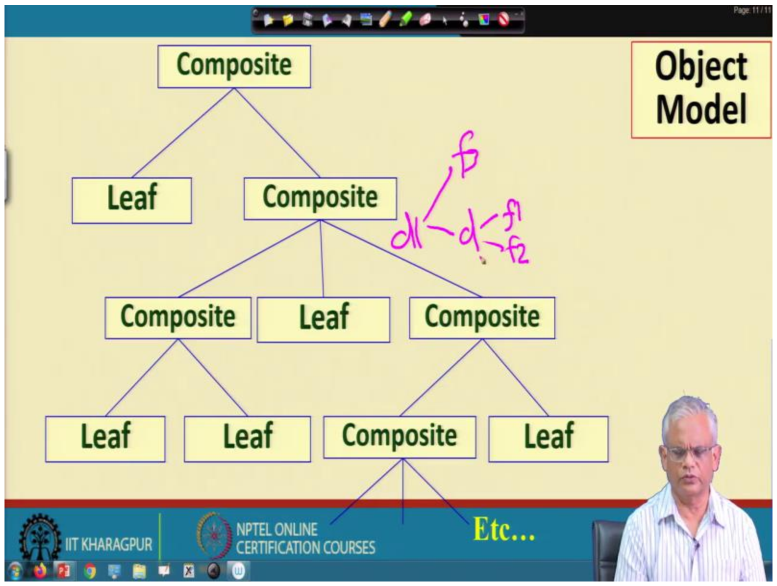
Again it is a composite pattern your folder is the interface and then it can contain primitive files or it can contain directories. The directory is the composite, folder is the component which is the interface, the directory contains many children some of the children can be directories or there can be primitive files.

(Refer Slide Time: 15:14)


Now Suppose file types are text, image, Video...
Component A
setName();
getName();
getAllFiles(List theList);
getContents(List theList); A
*
Directory
TextFile
ImageFile
VideoFile
setName();
getName();
getAllFiles(List theList);
getContents(List theList);
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

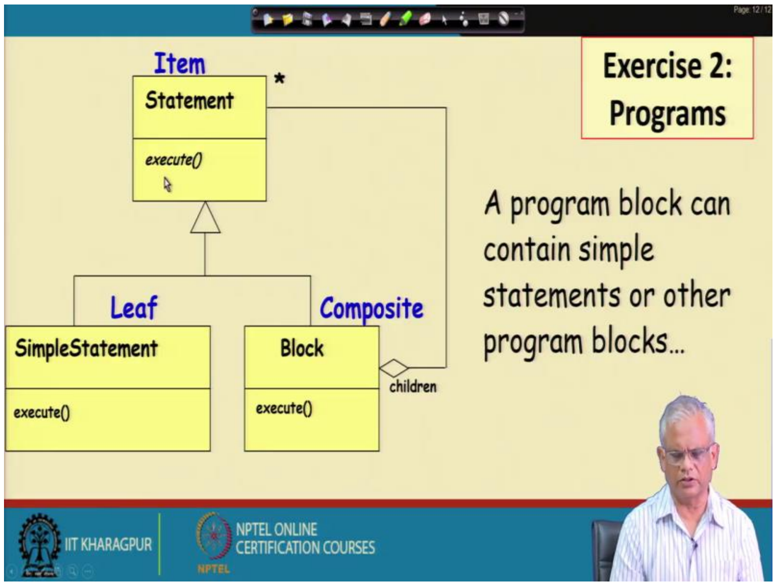
Now, let us say we want to change the problem little bit and say that some of the primitive files can be like text files, image files, video files and so on, so how do we change our design? So, this is again the composite pattern, here we have many times of primitive files, text file, image file, video file and so on and then the composite here that can contain many files some of them can be text file, image file, video file and also the directors.

(Refer Slide Time: 15:54)



But what will be the object model? The object model is that we start with the basic leaf level objects and see that how they are formed into composites, the larger composites and so on, it is a tree hierarchy form here. We can take a specific example, let us say we have files f1, f2 in a directory called as d and then d1 contains d and another file f3 and so on, so this is kind of the object diagram that what are the objects contained in the composite objects.

(Refer Slide Time: 16:58)



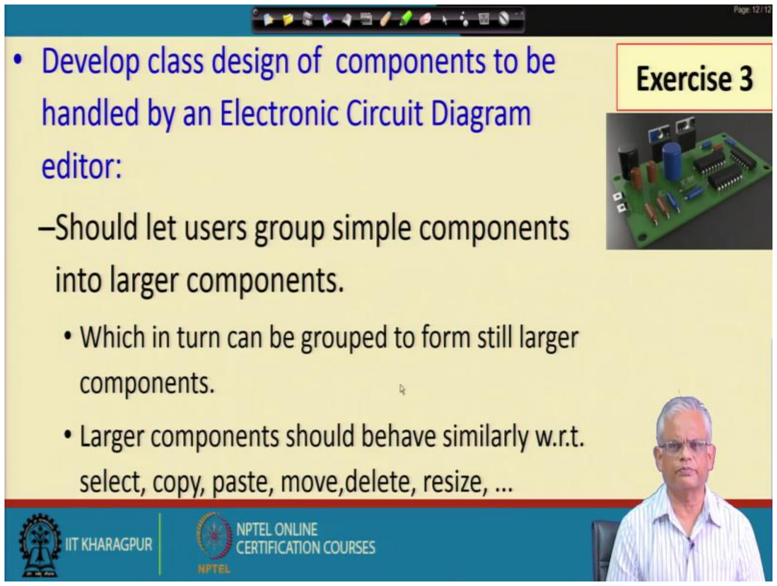
Exercise 2: Programs

A program block can contain simple statements or other program blocks...

Now, let us do one more exercise, let us say a program block can contain simple statements or other program blocks and then we can form larger program blocks by using these primitive program blocks, other statements and so on. So, what will be the class structure for this? Please try out.

If we know the composite pattern we can straight away apply here, the abstract class here or the interface class here is the statement or we can give some other name here if this name is not very we can call it as an element or something if that makes it more understandable because statement appears like a concrete one, make it more abstract, we might call it as a element, program element. And the program element can be a leaf element like a statement or can be a block, block contains many elements, each element can be statement or block.

(Refer Slide Time: 18:30)


- Develop class design of components to be handled by an Electronic Circuit Diagram editor:
  - –Should let users group simple components into larger components.
    - Which in turn can be grouped to form still larger components.
    - Larger components should behave similarly w.r.t. select, copy, paste, move,delete, resize, ...

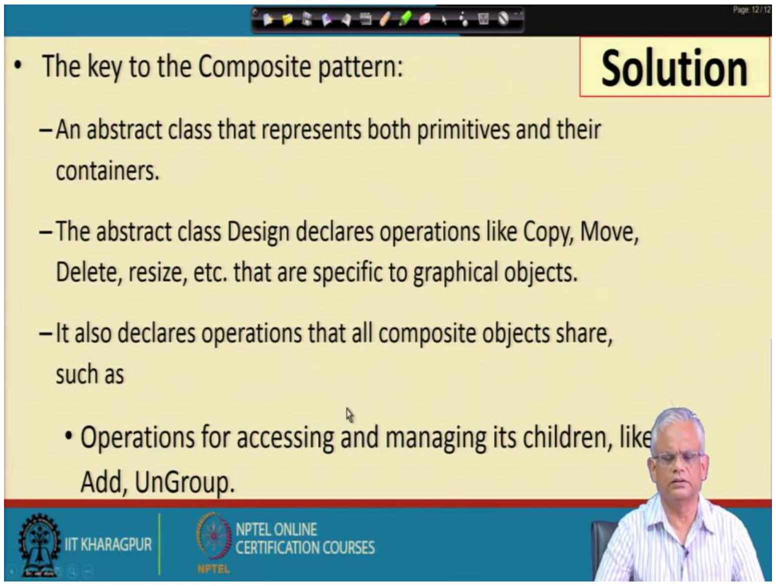
Exercise 3

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, let us do one more exercise, we want to develop a class design where we want to handle components, electronic components which are assembled into boards here and each board can contain many smaller boards and so on. Typically, an electronic circuit where we have some primitive circuits like capacitor register and so on, we form large components and that and so on.

And here our design should let designer form components using primitive components and still larger components using primitive components and some of the composite components and so on. You can realize that this is a straightforward application of the composite pattern if we know the composite pattern we can solve it effortlessly, please try to draw the class diagram for this problem.

(Refer Slide Time: 20:00)

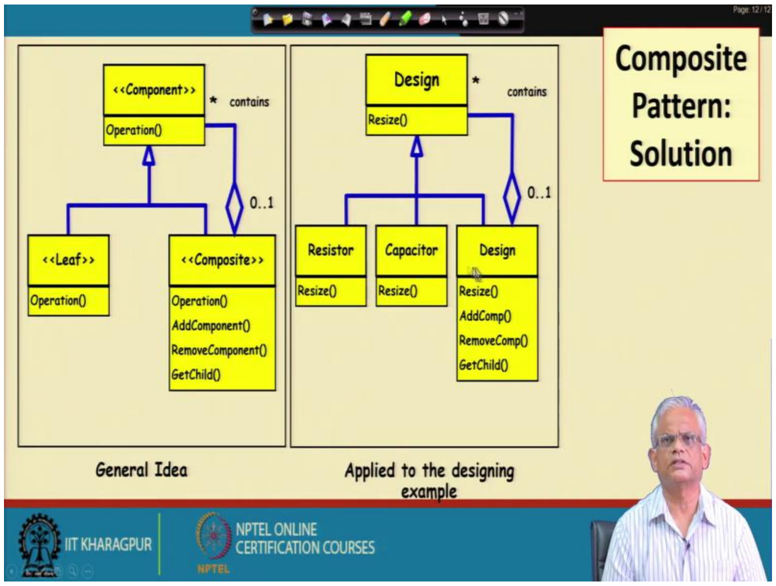


Solution

- The key to the Composite pattern:
  - An abstract class that represents both primitives and their containers.
  - The abstract class Design declares operations like Copy, Move, Delete, resize, etc. that are specific to graphical objects.
  - It also declares operations that all composite objects share, such as
    - Operations for accessing and managing its children, like Add, UnGroup.

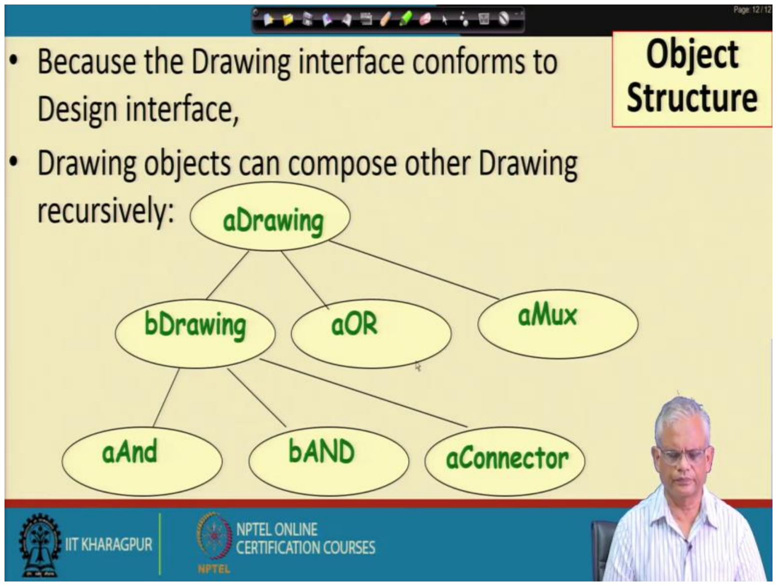
Here the solution, the key to the solution is that we have to define an abstract class that represents both primitive and the container and we will have operations like move, copy, resize, etc. that should be applicable to the primitive objects as well as to the composite objects and also there will be some operations for managing children like add, ungroup, etc.

(Refer Slide Time: 20:32)



So, this is the solution here, this is the general idea of the composite pattern and for this specific case, we will have the basic elements like register, capacitor, etc. and then one board design which contains many basic elements and other designs, other boards.

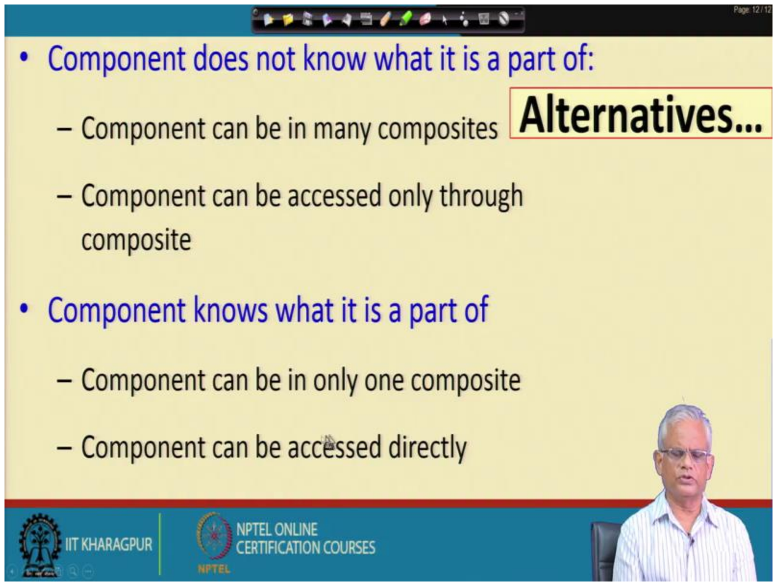
(Refer Slide Time: 20:58)



- Because the Drawing interface conforms to Design interface,
- Drawing objects can compose other Drawing recursively:

This is the object diagram where a composite; may contain only the primitive ones or it can contain some primitive ones and other composites.

(Refer Slide Time: 21:18)



- Component does not know what it is a part of:
  - Component can be in many composites
  - Component can be accessed only through composite
- Component knows what it is a part of
  - Component can be in only one composite
  - Component can be accessed directly

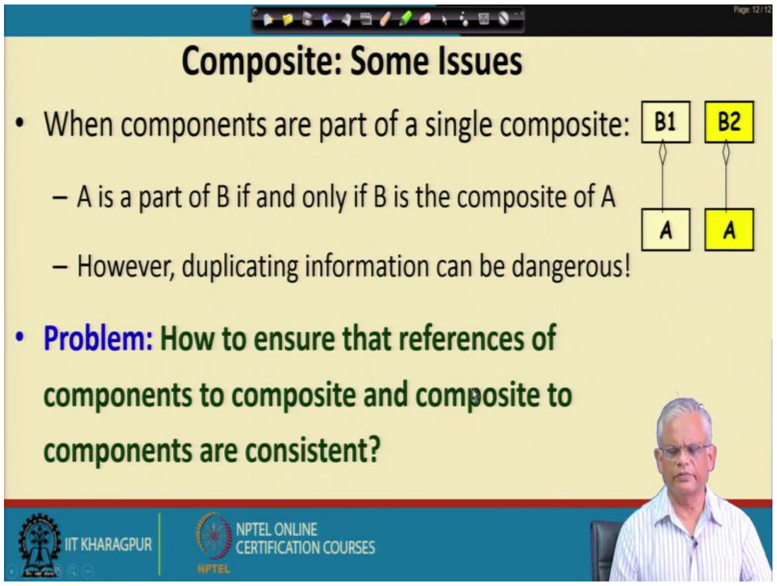
Alternatives...
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, there are some alternatives to this the way the solution is done for this pattern. One is that, once we form the group, a component does not know what it is part of. So, basically a component knows its children but not vice versa. The other one is that the component knows what it is part of, so the children also knows to which it belongs.

Then you might wonder that what is the advantage of the second one, why should a component or a basic element know to which it belongs, what would be the advantage of that? The advantage is that if the same element is part of many components we might save by repeating that across all the components because it may belong to multiple components.

So, here the component may be in multiple composites and it can be accessed only through the composite. The other case the component can be in only one composite and it can be accessed directly.

(Refer Slide Time: 22:52)

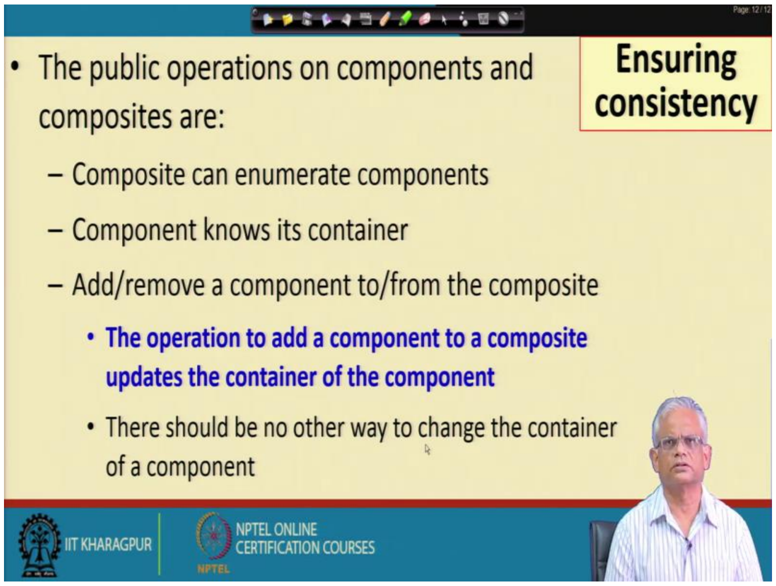


Composite: Some Issues

- When components are part of a single composite:
  - A is a part of B if and only if B is the composite of A
  - However, duplicating information can be dangerous!
- **Problem:** How to ensure that references of components to composite and composite to components are consistent?

So, when the components are part of a single composite, if A is part of B, if and only if B is composite of A, but then here we can have duplication of information and the problem will be that how to ensure that references to the components to composite and composite to component are consistent if we add composites, delete components, delete composites and so on it may become inconsistent.

(Refer Slide Time: 23:43)



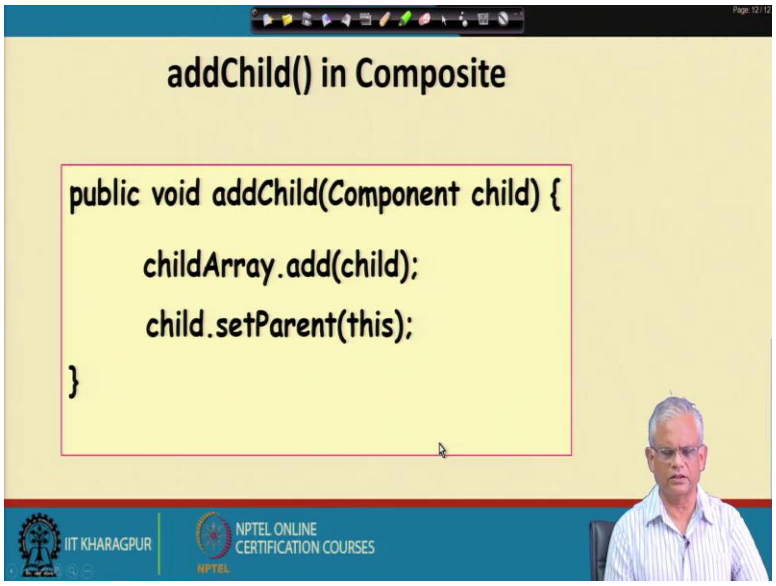
Ensuring consistency

- The public operations on components and composites are:
  - Composite can enumerate components
  - Component knows its container
  - Add/remove a component to/from the composite
    - **The operation to add a component to a composite updates the container of the component**
    - There should be no other way to change the container of a component

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

To ensure consistency, the composite should be able to enumerate components, the component knows its container and in the presence of add, remove operations to add a component to a composite we must update the container of the component and there should be no way to change the container of the component, just to ensure the consistency when we add a component we do the operations consistently that in the composite we add it and also in the child we are the to which it belongs, and when we delete we also do the same operation consistently.

(Refer Slide Time: 24:31)



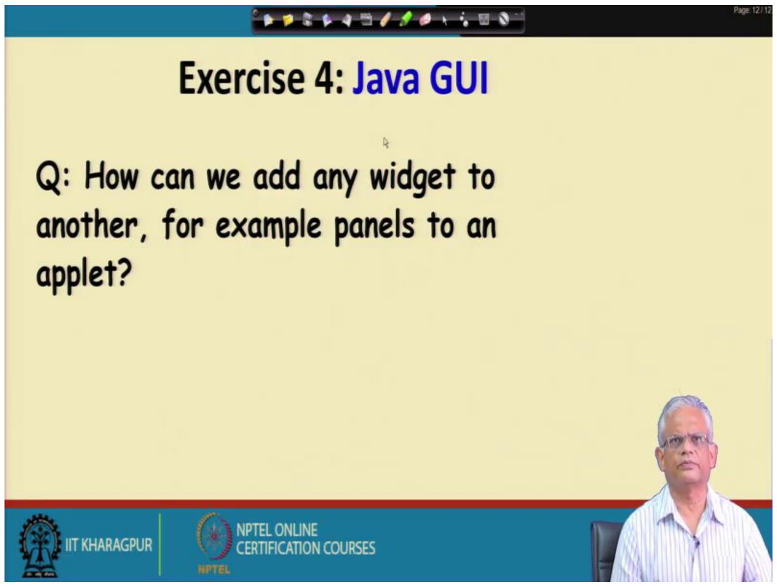
addChild() in Composite

```
public void addChild(Component child) {
        childArray.add(child);
        child.setParent(this);
}
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

So, this is the add child in the composite we not only add child here to the composite but also we set the parent for the child class.

(Refer Slide Time: 24:45)



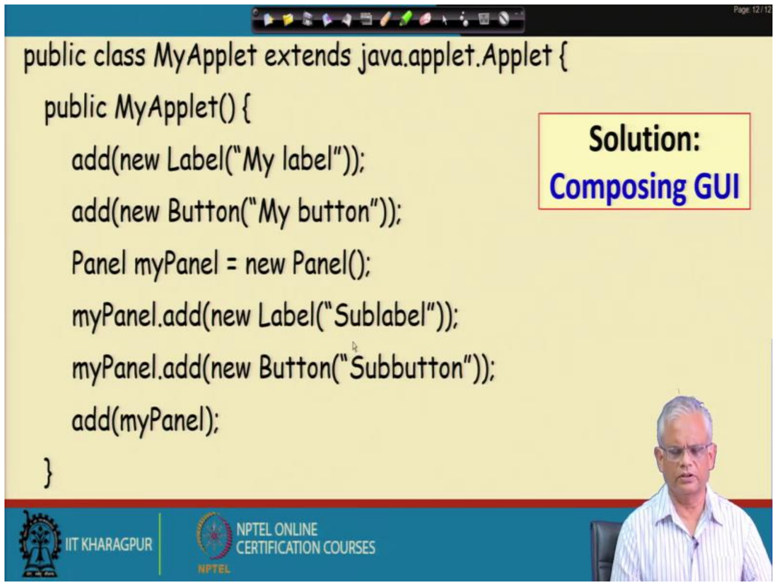
Exercise 4: Java GUI

Q: How can we add any widget to another, for example panels to an applet?

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Another example here about the Java GUI. The problem here is that how we can add a widget to another for example a panel to an applet?

(Refer Slide Time: 25:04)



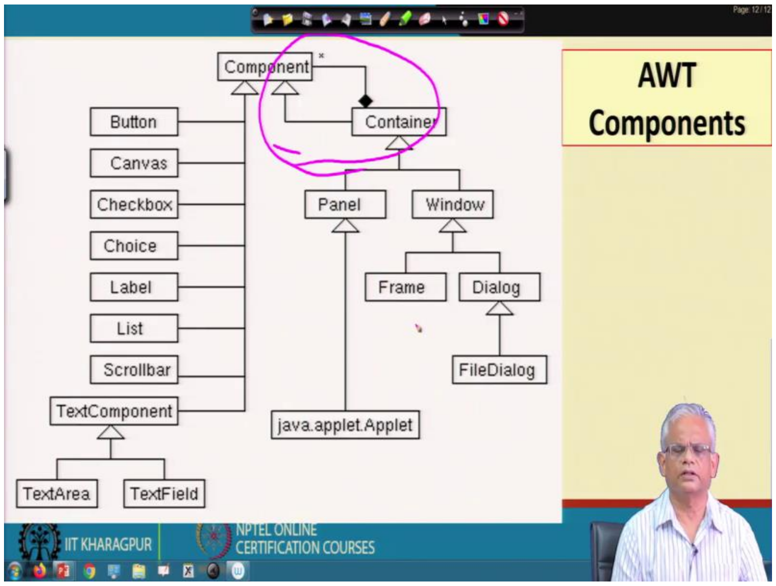
```
public class MyApplet extends java.applet.Applet {
  public MyApplet() {
    add(new Label("My label"));
    add(new Button("My button"));
    Panel myPanel = new Panel();
    myPanel.add(new Label("Sublabel"));
    myPanel.add(new Button("Subbutton"));
    add(myPanel);
  }
```

Solution: Composing GUI

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

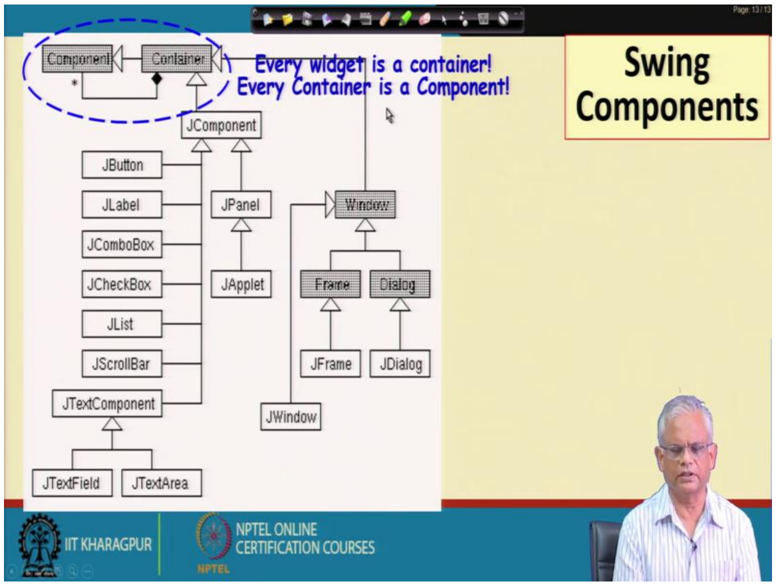
You might have done this kind of programming without knowing that you have been using the composite pattern. The applet, my applet extends the java applet and here you add some primitive elements like label, button and then on a panel this is a composite the panel you add sub label, sub button and then finally add the panel to the applet.

(Refer Slide Time: 25:38)



The Java GUI makes use of the composite pattern profusely just look at the AWT components here taken from the Java class diagram. So, see here that the component is an interface and there can be primitive components here like button, canvas, etc. or there can be composites here, just see here a direct application of the composite pattern here you can notice here, that there are primitive elements and then there is the composite here. And then each composite can be a window, a panel frame dialogue and so on.

(Refer Slide Time: 26:41)

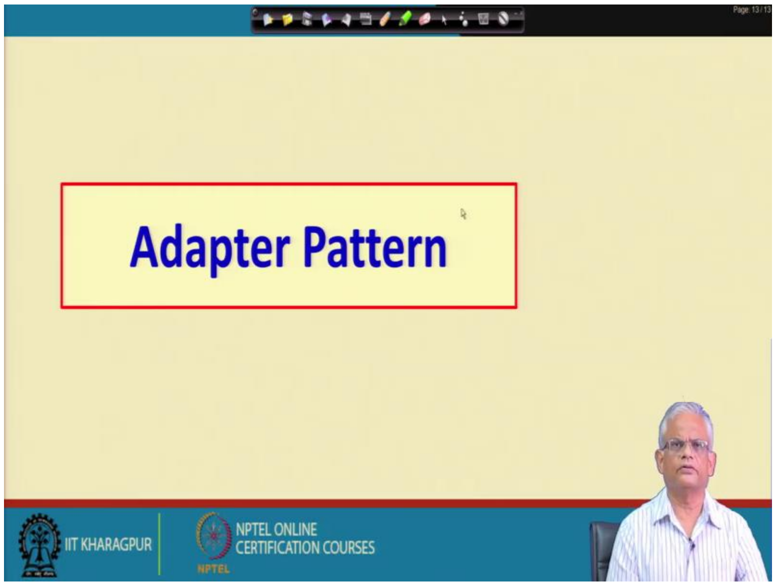


Swing Components

Every widget is a container!
Every Container is a Component!

The Java swing a small difference here, here these are also containers as you might know that in the Java swing; these are also containers the J button, J label and so on the small difference in the diagram here. So, here every widget is a container and every container is a component.

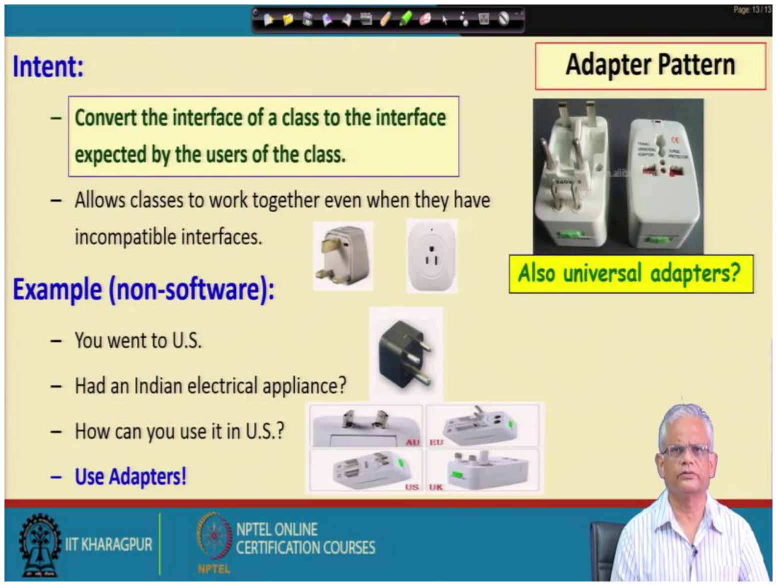
(Refer Slide Time: 27:17)



Adapter Pattern

So, we have completed our discussion on the composite pattern, now let us look at another very important pattern called as the adapter pattern.

(Refer Slide Time: 27:34)



Adapter Pattern

Intent:

- Convert the interface of a class to the interface expected by the users of the class.
- Allows classes to work together even when they have incompatible interfaces.

Example (non-software):

- You went to U.S.
- Had an Indian electrical appliance?
- How can you use it in U.S.?
- **Use Adapters!**

Also universal adapters?

The intent of this pattern is that how to convert the interface of a class to the interface expected by the users of the class. A class might have many users and they might use to calling some methods but the class under consideration has different methods. So, how do these incompatible classes work?

Let us look at an example which is a non-software example. Let us say you have been using some appliance, let us say mobile phone's charger and you went to USA and you had this charger, your charger in India is like this but there you find a socket like this, so how do you make your charger work in USA? You use adapters, so this is the adapter that fits into here and you can plug this into this and if you go to different countries UK and so on, the charger, the adapter for the USA will not work in UK, there you need a different type of adapter so you might have a universal adapter which fits into the socket of any country you visit.

The same thing happens in software; a class invokes methods of another class but then you want to use a different class which has different set of methods. So how do you make these two classes work? So, that is about the adapter pattern we are almost at the end of this lecture, we will continue from this point in the next class. Thank you.