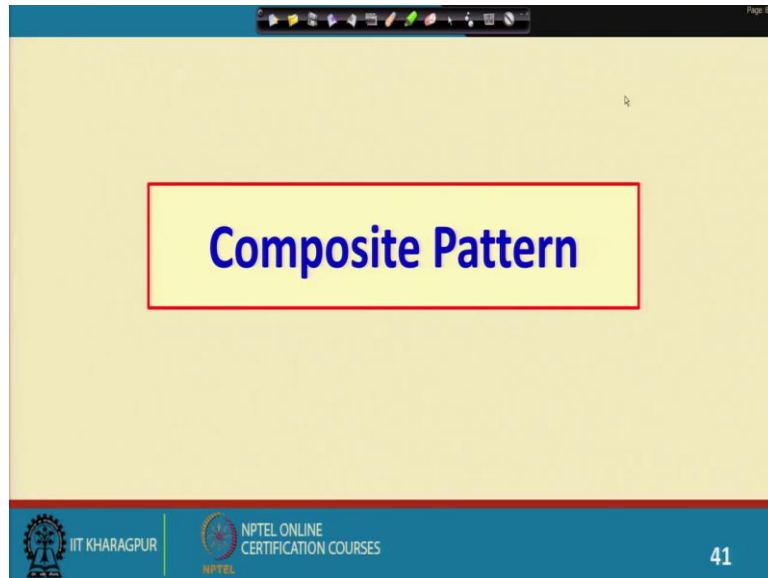


Object Oriented System Development Using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 49
Composite Pattern-I

(Refer Slide Time: 00:25)



Welcome to this session, in this session we shall be discussing about the composite pattern.

(Refer Slide Time: 00:30)

Composite: Introduction

- A composite is a group of objects in which some objects contain others:

- An object may represent a group.
- Or may represent an individual item.

```
graph TD; t1["t1 : Group"] --> a["a : Leaf"]; t1 --> b["b : Leaf"]; t1 --> t2["t2 : Group"]; t1 --> c["c : Leaf"]; t2 --> t3["t3 : Group"]; t2 --> e1["e : Leaf"]; t3 --> d["d : Leaf"]; t3 --> e2["e : Leaf"];
```

Now, let us get started. Many times, while solving an application we find that there are groups of objects which contain other primitive objects and groups of object, it is a very common problem. Just to give an example, that you might be having a drawing package, in

the drawing package you may draw lines, you may draw circles, rectangles, and then you may select this and make this into a group.

And then you can deal with the group just like a primitive element, for example you might copy the group and the group will appear here, you might draw few more primitive objects and then again you may form a group which contains two groups and a primitive object, and then move it that will just like primitive object is moved you can move a group, you can reduce the size or expand the size, you can copy it and so on.

This is the very common problem will see that in many places it occurs, basically if you look at here a group contains some leaf elements and also can contain a group and this group in turn contains another group here and another leaf here, and this group contains only primitive elements. In the drawing example we first draw the primitive element for one group. And then add more primitive elements and then form a larger group and then add more elements or we might repeat this group here and all these will appear here and then we form still a bigger group and so on. This is a very common problem in many applications. So, here an object may represent a group or it may represent a leaf level item or an individual item. But then they must behave similarly.

How do we structure the classes here, that is the main problem, it is a structural pattern where we are interested in determining how the different classes participating in the group, some of them are primitive, some of them are composite how will they be structured? So, that is the main problem that we are trying to address here.

(Refer Slide Time: 03:59)

Composite: Introduction

- A composite is a group of objects in which some objects contain others:
- An object may represent a group.
- Or may represent an individual item.
- Example: A CAD Design ---

The diagram illustrates a composite structure. At the top is a purple box labeled 't1 : Group'. It has four children: 'a : Leaf' (yellow), 'b : Leaf' (yellow), 't2 : Group' (purple), and 'c : Leaf' (yellow). 't2 : Group' has two children: 't3 : Group' (purple) and 'e : Leaf' (yellow). 't3 : Group' has two children: 'd : Leaf' (yellow) and 'e : Leaf' (yellow). The slide also features a small inset image of a CAD design and a video feed of a presenter.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Another example is a Computer Added Design package, you can draw a very sophisticated design here a layout of a VLSI and then here you form many groups and the groups are repeated changed and so on. And it makes it easy to draw just because you are using the notion of nesting objects, forming groups and so on.

(Refer Slide Time: 04:36)

Example: Consider a CAD Editor...

- You can build complex diagrams using simple components
- Group components to form larger components...
- ...which in turn can be grouped to form still larger components

The screenshot shows a CAD editor window with a grid. A complex diagram is being built with components labeled 'Bridge', 'Slide A', and 'Slide B'. A red arrow points from a 'Group' button in the software's menu to the diagram. A blue arrow points from a 'Resize Shape' button to a specific component in the diagram. The slide also features a video feed of a presenter.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

There are other examples, this is a mechanical CAD, mechanical Computer Added Design editor. You can build complex diagrams using simple components. For example, you select the side A, side B and bridge and then this becomes a group. Now, you can have, you can copy it and paste it, add more primitive elements from larger groups and so on.

So, here just using the drawing package we selected copy paste here, we got the same one here the group is pasted, we can resize and so on. We the group we pasted here resized the group and still made we can make a larger group and so on. So, we again paste it the group we resized and then we copied it and again paste it, so these are some of the common things that are drawn in an editor. Now, let us our focus here is that how do we use the composite pattern, what is a good solution for this problem.

(Refer Slide Time: 06:29)

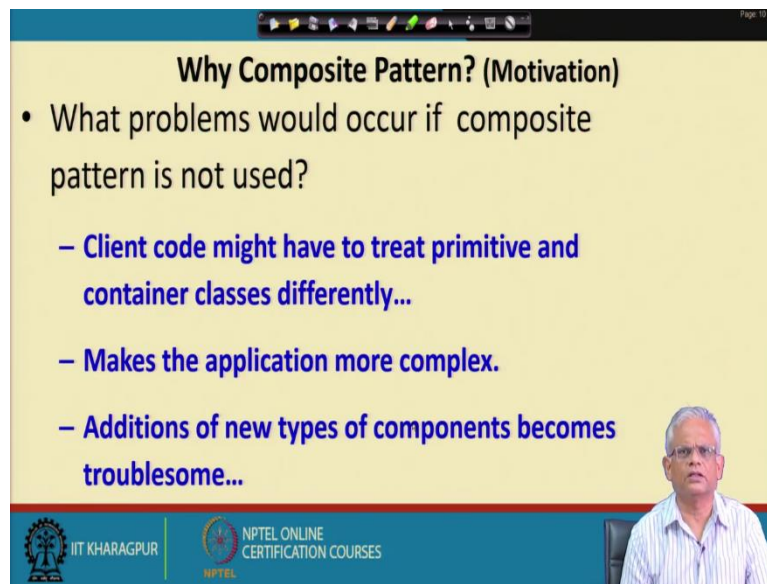
The slide is titled "Composite Pattern: Intent". It features a yellow background with a red border around the main text area. On the right side, there is a tree diagram illustrating the Composite Pattern. The root node is a pink box labeled "t1_Group". It has four children: "e_Leaf", "e_Leaf", "t2_Group", and "e_Leaf". The "t2_Group" node has two children: "t3_Group" and "e_Leaf". The "t3_Group" node has two children: "e_Leaf" and "e_Leaf".

- Compose nested groups of objects into a tree structure to represent part-whole hierarchies.
 - Clients should be able to treat individual objects and composites in the same way.

The slide also includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES at the bottom. A presenter is visible in the bottom right corner of the slide.

The intent of the composite pattern is to compose a nested group of objects into a tree structure. A nested group of objects into a tree structured that represent part -whole hierarchies group consist of some primitive elements and another group here consist of primitive element and another group the top-level group here contains pre primitive elements and a large group. And one thing we must keep in mind that in these applications the user must not be able to do different things when dealing with a group or a simple element, a primitive element we can copy paste you can move, you can resize and so on, and the same thing we must be able to do using similar operations on a composite item, so that is the requirement here.

(Refer Slide Time: 07:50)



Why Composite Pattern? (Motivation)

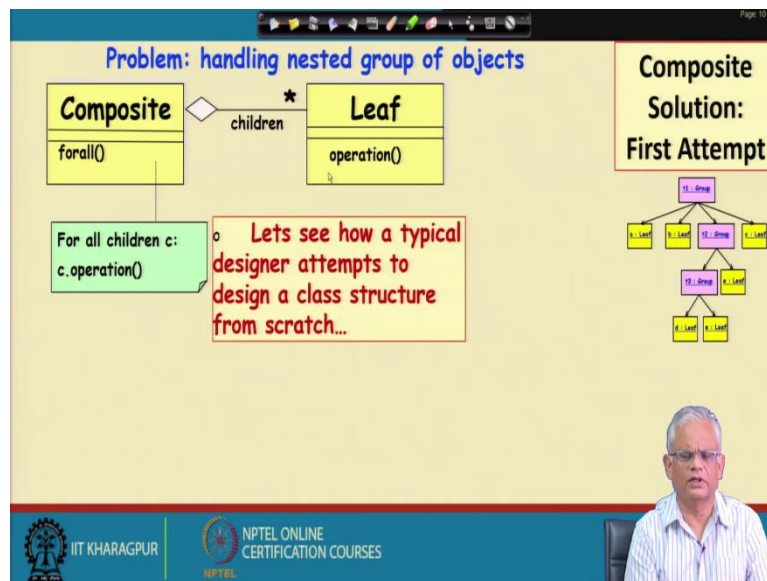
- What problems would occur if composite pattern is not used?
 - Client code might have to treat primitive and container classes differently...
 - Makes the application more complex.
 - Additions of new types of components becomes troublesome...

The slide includes a video inset of a man in a striped shirt speaking. At the bottom, there are logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

Now, first let see that if you do not use the composite pattern what would happen? If you do not use the composite pattern one is that the user or the client might have to treat the primitive and the composite differently, for example if it is a primitive element we might just move it if it is a composite we might have to move repeat the move operation across all the elements and so on that will be not a very elegant or acceptable solution.

The application would become complex, and if we add new types of primitive elements and new groups it would become extremely complicated troublesome for the user. And as you can see that the composite pattern solves a very important problem and in the right situation you would have no difficulty, in identifying the right situation and using the composite pattern which will really help in making your application elegant.

(Refer Slide Time: 09:16)



Now, let see if we pose this problem to a novice programmer that see we have this group of elements which would be able to form group of elements using primitive elements and then we can form larger groups containing groups and primitive elements and so on and the group and the primitive element must behave similarly with respect to copy, paste, resize, move, delete and so on.

Now, how will a programmer approach this problem, let see that the mistakes he will do so that it will become clear to us that the elegance of the composite patterns, if we pose this problem to a novice the first thing that will strike to his mind is that there is a nested group of objects. And how does he do the nesting. So, might so this be the nested group of objects and he will try to address this problem and a typical designer who does not know the composite pattern, he will come up with this kind of solution.

A composite contains many leaves and then the composite method supported is for all and here for the leaf it is operation. And if you want to let say move a composite it will invoke the for all and the operation specific operation may be move as just written here operation, the operation may be move, may be delete, may be resize etc. So, the composite client would have to call for all and then that will intern call the corresponding operation and the leaf.

But if the client deals with a primitive object you would have to call the operation directly. Now let see what are the difficulties with these solutions, as you can imagine the first difficultly is that the client has to treat the composite and the leaf separately, he has to invoke for all and the composite and then the leaf he has to directly call the operation.

And another thing to notice here is that the single level of hierarchy, what we really needed was a nested tree and also the composite contains one type of leaf, we wanted it to contain many other composites and leaf, definitely not a very good solution.

(Refer Slide Time: 12:45)

Problem: handling nested group of objects

Composite (class): forall()

Leaf (class): operation()

Composite o-- "*" children --> Leaf

For all children c:
c.operation()

Mark=1/100

Analysis of Solution: Naïve solution...

What are the problems?

- Only single nesting level (depth =1)
- Composite and leaves always treated differently, Difficult to extend.

Client Code:

```
...  
if X is Composite  
then  
    X.forall()  
else  
    X.operation();  
...
```

Composite Solution: First Attempt

```
graph TD  
    A["X.Composite"] --> B["a.Leaf"]  
    A --> C["b.Leaf"]  
    A --> D["X.Composite"]  
    A --> E["c.Leaf"]  
    D --> F["d.Leaf"]  
    D --> G["e.Leaf"]
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

It falls way short of what is required, it is a very naïve solution; the problems are that only a single level of nesting, the composite and leaves are treated differently, difficult to extend for example how do we add new composites, new leaves and so on, not a good solution. And if somebody is posed this problem as an examination question and he gives this answer then the examiner would possibly award him 1 out of 100 as not a good solution.

You can see here the client has to distinguish between whether he is dealing with a composite or a primitive operation, if it is a composite then it has to call the for all, otherwise he has to call the operation directly. Getting a complex group out of this is very difficult, nesting is not supported different types of primitives different composites is very hard to support not a good solution. Now, let say we ask the designer to back to the design table and to come up with a better solution.

(Refer Slide Time: 14:23)

The slide displays a class diagram for an **Item** class. It has a **children** list (indicated by an asterisk) and an **operation()** method. A note next to the operation method reads: "if there are children then for all children c: c.operation() else doSomething();". To the right, it says "Mark= 40/100" and "Composite: Attempt 2". Below the diagram, there is a list of problems:

- Does not handle different item types: primitive and composite
- Difficult to extend with new kinds of leafs or composites.

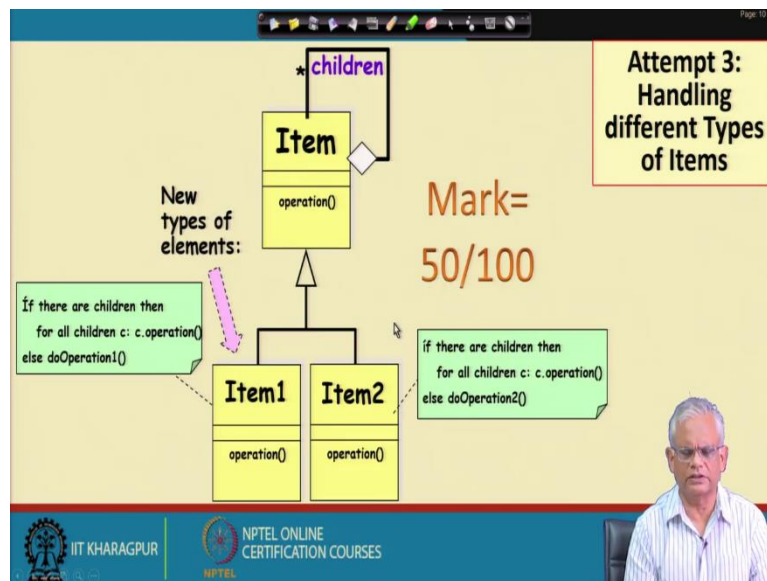
The slide also features the IIT Kharagpur and NPTEL logos at the bottom.

The second attempt tries to address the problem that there was no hierarchy, there was only a single level of hierarchy. He might come up with these that an item contains many children and the item can have further children and so on, you can see that there is a multiple level here possible. And here in the item can be a composite or it can be a primitive element and if it is a composite, if there are children then for all children do the operation else do something.

Again, not a very elegant solution because how do we support new types of primitive elements, new types of composites and also if then else does not look okay, definitely there are improvement over the previous attempt as the previous attempt was restricted to a single level of hierarchy and here we are having multiple levels and also kind of unified treatment for the composite and the primitive element the client does not have to really distinguish between the composite and the primitive element he just makes a method call an operation and then the operation internally checks whether there are children then do the operation for all the children otherwise do something.

But then what are the problems? The problems are that does not handle different item types, different primitives and different types of composites, difficult to extend. Now we have to ask the designer to go back to the design table and come with a better solution but if we as an examiner we have to give a mark to this the extent to his it satisfies the requirement of the original problem we might give 40 out of 100.

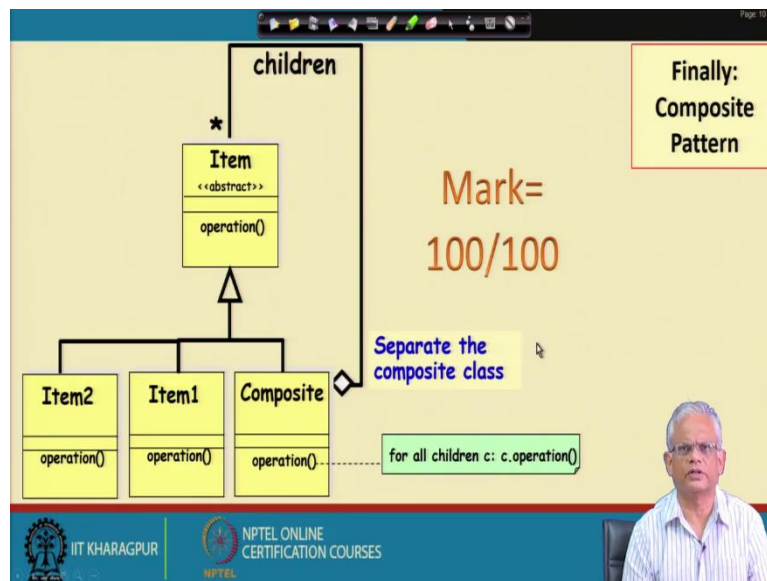
(Refer Slide Time: 17:11)



Now, the attempt 3; the designer comes up with this design. Extended basically the previous solution there is hierarchy here the item and then there are different types of items supported here through inheritance relationship, the item can be item 1, item 2 and so on. Again, this is not a good solution definitely improvement over the previous one because different types of item can be supported primitive different types of primitive and composite can be supported.

But then, one is that item is a concrete class and then these are all concrete classes and then we will have difficulty with respect to the list of substitution principle and also to group items into multiple primitive elements and composite elements does not look very correct, we have been able to add new types of elements that is the improvements and if we have to give them mark, will say that it is 50 out of 100, so this is the third attempt and the designer.

(Refer Slide Time: 18:52)

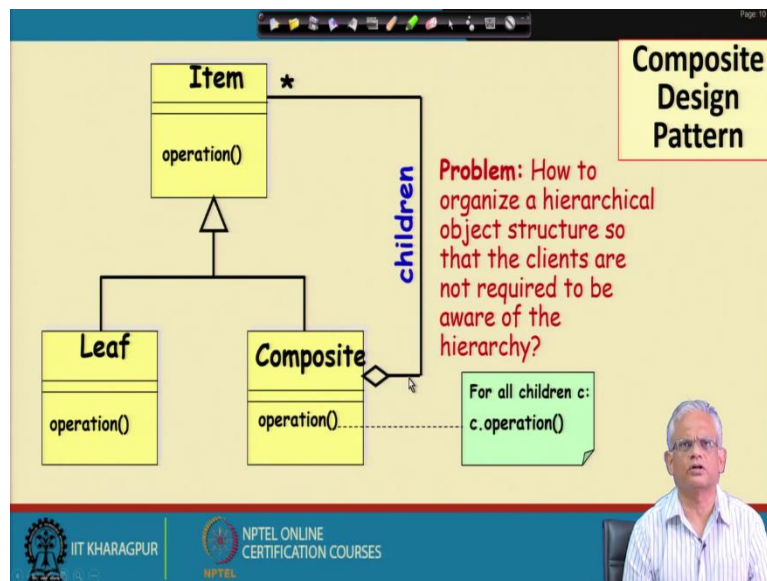


Now, we ask the designer to go back to the design table and come with a better solution, now he comes up with a solution that an item is abstract here does not violate the basic principles, object-oriented principles like the Liskov substitution principle and then the derived classes are either primitive elements or a composite, there can be different composites of course.

And then the composites contain the items which can intern be primitive items or may be composites themselves. And this is the solution that we required and this exactly is called as the composite pattern. Here an item can contain many primitive items or it can contain some primitive items and some composites and so on, so this is the solution that we required and this is called as the composite pattern as you can see.

If you do not know the solution you will have to really struggle to come up with this solution and that is why it is good to learn this composite patterns well and apply it wherever required to come up with elegant object-oriented programs.

(Refer Slide Time: 20:41)



Now, this is the pattern just redrawn that there can be multiple leaves and multiple composites. I just simplified representation of the composite design pattern. The composite contains many children and each child can be a leaf object or a composite object. And the problem the composite pattern addresses as we have already said that how to organize a hierarchical objects structure.

So, that the clients are not required to be aware of the hierarchy, the deal with large composites small composites, primitive objects exactly in the same way, and for all as you can see that they just invoke the specific operation, without having to know whether it is large composites small composite or a leaf.

(Refer Slide Time: 21:51)

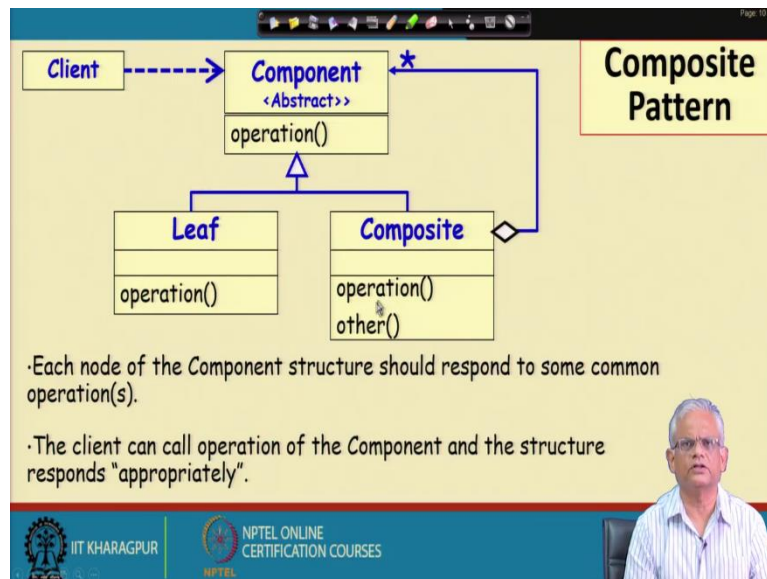
The slide is titled "Composite Pattern: Issues" and lists four key questions:

- What is the class diagram?
- How does the client interact?
- What operations are defined for:
 - The component, the composite, and the leaf?
 - How are they carried out?
- How is the design implemented?

The slide includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

Now, let us look at the issues here. We must be able to understand and apply the composite pattern we must know the class diagram which I have already seen, we must know how the client interacts with a composite object and the primitive objects. What are the operations that are defined for the component the composite and the leaf? How are they carried out and how is the design implemented that is what will be the Java code look like for a given class structure.

(Refer Slide Time: 22:35)

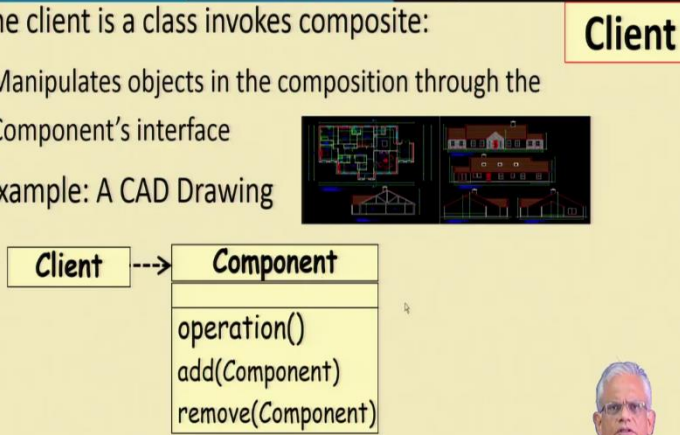


Now let us address those issues, the first is class diagram. This is a very elegant solution and if we say that composite pattern this must come immediately to the mind that there is abstract component and then there are primitive leaf elements and then there are composites. The composite contains many components and each component can be a leaf for a composite. The client invokes the operation and the component and the component may be either a leaf for a composite. And the operation is defined here in abstract class and necessary over written here.

(Refer Slide Time: 23:27)

• The client is a class invokes composite:

- Manipulates objects in the composition through the Component's interface
- Example: A CAD Drawing



Client

Component

operation()
add(Component)
remove(Component)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

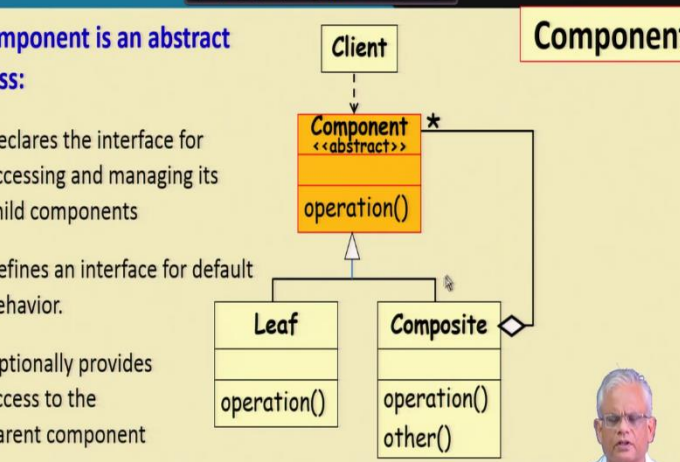
Now, let us look at how the client interacts with the composite pattern. The client manipulates the composite through the component interface, component is the abstract class and the different methods that are defined and the component it manipulates the different objects, the object can be either a primitive object or a composite object.

So, these are the operations the operation add remove and the client evokes these without knowing whether is doing dealing with a simple primitive object a small group or very large group, the same operations are supported by all. So, this typically happens in a CAD drawing there are many other applications as we will see just now.

(Refer Slide Time: 24:30)

• **Component is an abstract class:**

- Declares the interface for accessing and managing its child components
- Defines an interface for default behavior.
- Optionally provides access to the parent component



Client

Component <<abstract>> *

operation()

Leaf

operation()

Composite

operation()
other()

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Here, please remember that the component is abstract class it declares the interface which will be usable by the client, it defines some behavior that are common to leaf and composite and the composite may define more methods will see these methods.

(Refer Slide Time: 25:08)

- **Leaf**
 - A leaf has no children.
 - Defines behavior for primitive objects in the composition.
- **Composite**
 - Defines behavior for components having children.
 - Stores child components.
 - Implements child-related operations in the Component interface.

Other Participants

```

classDiagram
    class Client
    class Component {
        <<abstract>>
        operation()
    }
    class Leaf {
        operation()
    }
    class Composite {
        operation()
        other()
    }
    Client --> "*" Component
    Component <|-- Leaf
    Component <|-- Composite
    Composite o-- "*" Component
    
```

Now, the leaf has no children and it just defines the operation that is defined in the component. The composite in the addition to the operation define and the component must have operation for child management. For example, add child, delete child and so on. So, the composite defines behavior for the components children store or add child component and other child related operations invoke different operations and the child delete etc.

(Refer Slide Time: 25:55)

- Clients use the Component class interface, which in turn interacts with objects.
- If the recipient is a Leaf:
 - **Handles the request directly...**
- If the recipient is a Composite:
 - **Forwards the request to its child components...**

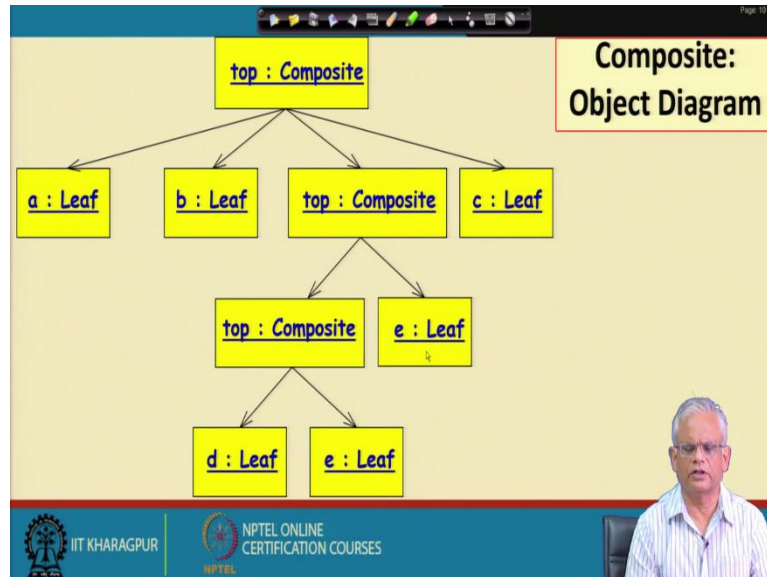
Collaborations

```

classDiagram
    class Client
    class Component {
        <<abstract>>
        operation()
    }
    class Leaf {
        operation()
    }
    class Composite {
        operation()
        other()
    }
    Client --> "*" Component
    Component <|-- Leaf
    Component <|-- Composite
    Composite o-- "*" Component
    
```

Now, the client interacts using the component interface and the component may be a leaf composite if the recipient the client method invocation is a leaf object then it directly handles the query if it is composite then it just invokes the corresponding operations on the children objects.

(Refer Slide Time: 26:30)



And this is the object diagram, different objects here there are composite objects which contain other objects and primitive objects. So, it will be note very difficult for you to draw the object diagram given a code or a running of a code.

(Refer Slide Time: 26:55)

The slide is titled "Consequences" and features a yellow box with the following text: "Makes it possible to define recursive composition of primitive and composite objects." Below this, there are three bullet points:

- Makes invocations by client simpler.
 - Client doesn't need to know whether it is dealing with leaves or composites.
- Makes it easier to add new kinds of components.

 To the right of the text is a UML class diagram showing a "Client" class with an association to an abstract "Component" class (labeled "Component <<abstract>>") which has an "operation()" method. "Component" is abstracted by "Leaf" and "Composite" classes. "Leaf" has an "operation()" method, and "Composite" has "operation()" and "other()" methods. The slide footer includes the IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES logos.

This patterns it becomes possible to define recursive composition of the primitive and composite objects that was what was required to form the groups. And the client side the

client invocation is very simple does not have to distinguish between the composites and the primitives from the programming point of view one can easily add new kinds of primitives and composites.

(Refer Slide Time: 27:32)

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.**

The diagram shows a tree structure with a root node labeled 'S1_Group' (purple) and several leaf nodes labeled 'e_Leaf' (yellow). The tree structure is as follows: S1_Group has three children: e_Leaf, S1_Group, and e_Leaf. The middle S1_Group node has two children: e_Leaf and e_Leaf. The bottom S1_Group node has two children: e_Leaf and e_Leaf.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

This pattern is applicable wherever there is a part whole relation between objects but then we must be able to form larger groups from dynamically using the primitive elements and other groups, so this is very important that this is applicable when there are part whole hierarchies but then we must be careful that we must have the problem that involves creating composites using primitives and other composites dynamically at the run time. We will see some examples where will again highlight this point.

This is what is important not only the part whole hierarchies exist, but also the parts the composites should be created dynamically at run time we should be able to identify some elements and then form that into larger groups and then still form larger bigger groups and so on.

We can copy a group multiple times and make that into a larger group with more primitives and so on. We will take some examples and look at the kind of code that will correspond to these design pattern but we are almost at the end of this lecture, we will stop here and continue in the next lecture, thank you.