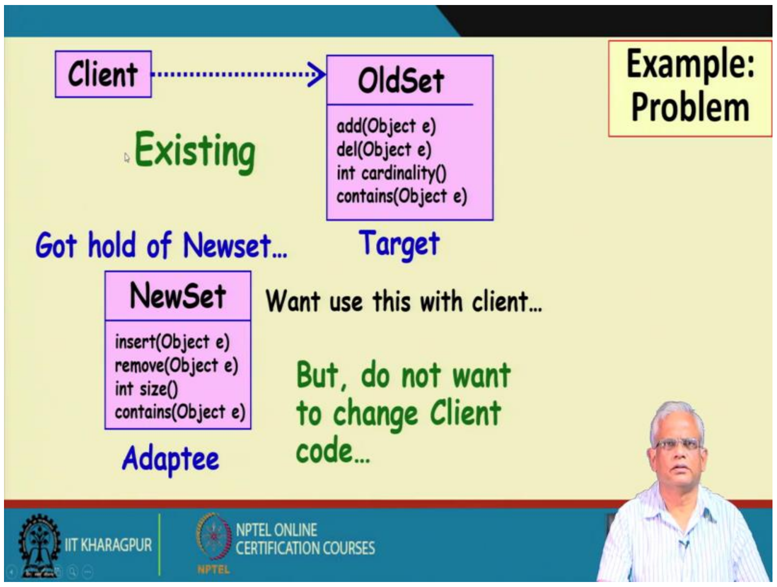
Object- Oriented System Development Using UML, Java and Patterns
Professor. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture No. 52
Adapter Pattern - 2

Welcome to this lecture! In the last lecture, we were discussing about the Adapter Pattern. We said that there are two main types of adapters. One is the class adapter and the other is object adapter. In the last class, last lecture, we are discussed about the object adapter pattern with the help of an example of a set and what will be the class structure for the adapter and the working of the adapter, even the Java code we had seen in the last lecture.

(Refer Slide Time: 01:06)



Example: Problem

Client → OldSet

OldSet
- add(Object e)
- del(Object e)
- int cardinality()
- contains(Object e)

Existing

Target

Got hold of Newset...

NewSet
- insert(Object e)
- remove(Object e)
- int size()
- contains(Object e)

Adaptee

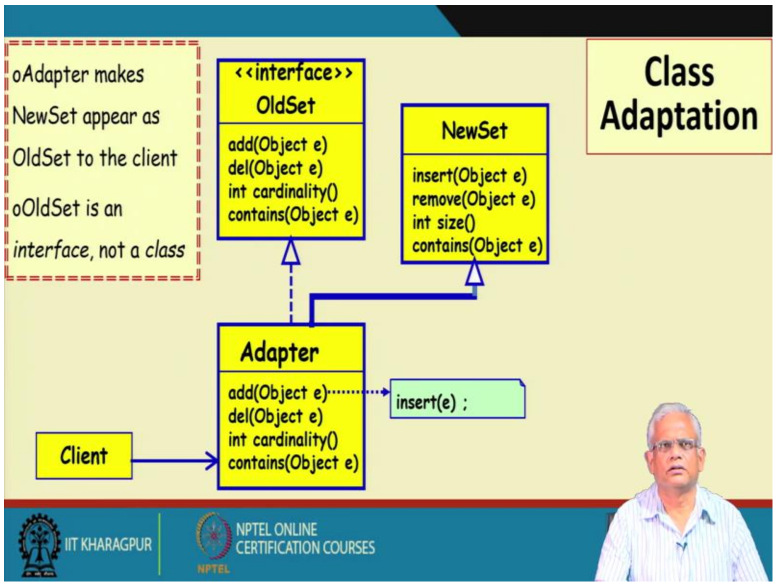
Want use this with client...

But, do not want to change Client code...

Now, let us look at the same example, about a set adapter using the class adapter pattern. The problem that we are discussing is that we had an old set and this is the old set, which is the target. The target interface is add, delete, cardinality and contains. But we got hold of a new set where, which is the adaptee, the new set is the adapter and methods supported are insert, remove, size and contains.

Now we want the client to use the new set without any change to the client code. We had seen how to achieve this, using the object adapter pattern in the last lecture. Now, let us see how to achieve the same thing using the class adapter.

(Refer Slide Time: 02:21)

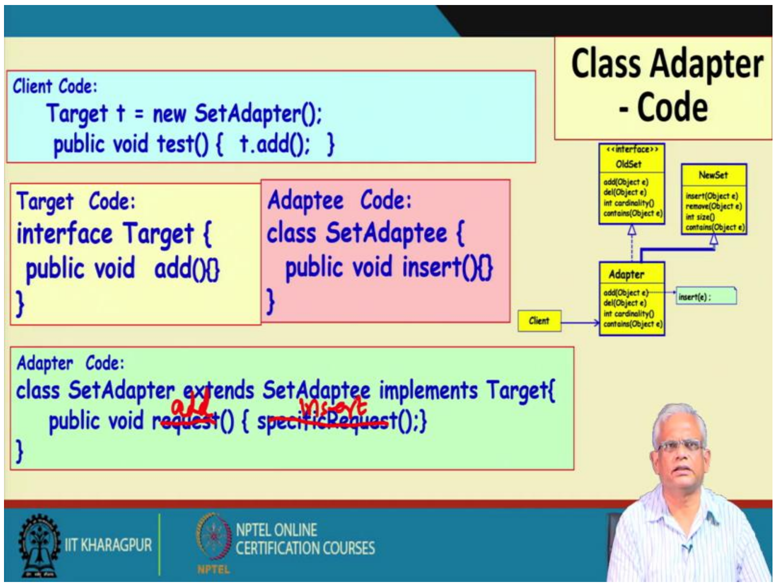


Class Adaptation

- Adapter makes NewSet appear as OldSet to the client
- OldSet is an interface, not a class

In the class adaptation, we have to write the adapter which implements the target interface, which is the old set interface. So, it should have the methods add, delete, cardinality and contains and it is a subclass of the new set, which is the adaptee. Being a subclass of new set, it has internally insert, remove, size and contains available due to inheritance and therefore, when the client makes a call, let us say add the adapter internally we will call insert, because the insert method is available due to inheritance. So, that is the main idea in the class adaptation. Now let us see the Java code for this.

(Refer Slide Time: 03:32)



Class Adapter - Code

Client Code:
```
    Target t = new SetAdapter();
     public void test() {  t.add();  }
```

Target Code:
```
interface Target {
  public void  add(){}
}
```

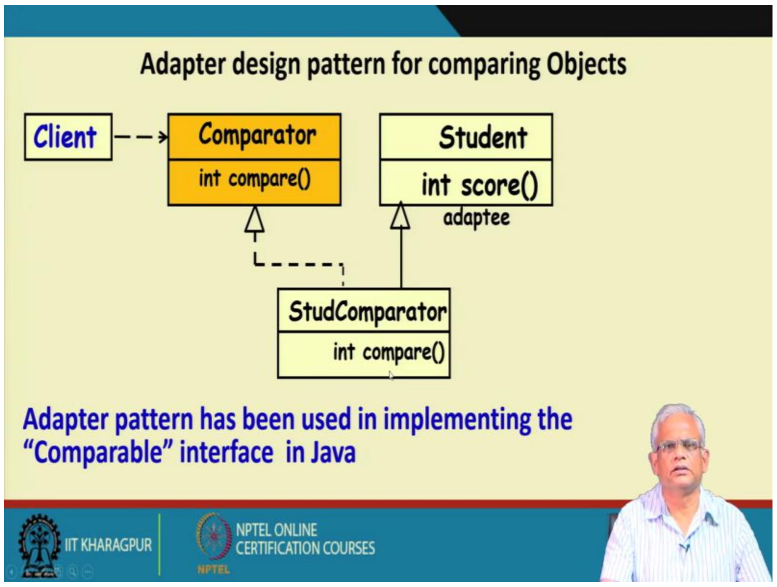
Adaptee Code:
```
class SetAdaptee {
   public void insert(){}
}
```

Adapter Code:
```
class SetAdapter extends SetAdaptee implements Target{
     public void ~~request~~ add() { ~~specificRequest~~ insert();}
}
```

Here, the client code, the client creates the set adapter and then we want to test it. So, t.add and need to have the new set which is the insert method invoked by the adapter. Now let us see the code for the target, which has the add method, I have not written the methods for all of this for simplicity, just considered the add interface target. We have the add method here and then the adaptee, this the adaptee and here we have the insert method, not written the body of the method just for our understanding, just try to simplify it and then we write the adapter code here.

The adapter, extends adaptee and implements target and we need to provide in the implementation of the methods of the target, so here it said be add. So, whenever we have the add method here, because that is the implementation; the target is implemented. So, whenever the add is called here, we will call here insert. The class adapter and object adapter very simple. The class adapter implements the target and extends the adaptee and internally, whenever there is a call on the target interface, it just makes the call to the corresponding adaptee interface.

(Refer Slide Time: 06:22)



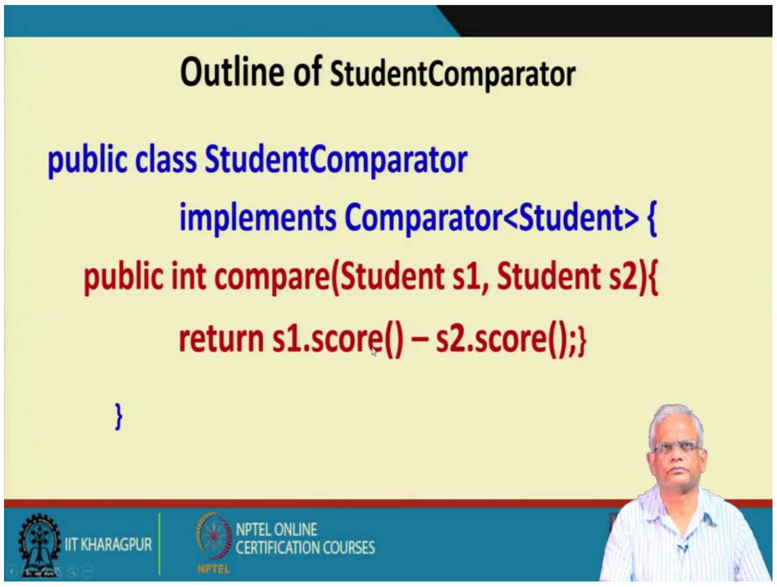
Adapter design pattern for comparing Objects

Adapter pattern has been used in implementing the "Comparable" interface in Java

Now, let us see some applications of the adapter pattern. One application of the adapter pattern, which almost everybody doing a Java programme needs to do some time or other, is to use a comparator class. The comparator is used when we have let us say, a number of students may be in a ArrayList or something and each student has a score and we want to compare the scores of the students to be able to let us sort the students in the ascending or descending order and so on.

So, we need to, the client needs to call a competitor, student competitor, which is essentially an adapter which implements the comparator interface and inherits the student and internally, it needs to provide a definition for the compare and the definition is given here.

(Refer Slide Time: 07:43)



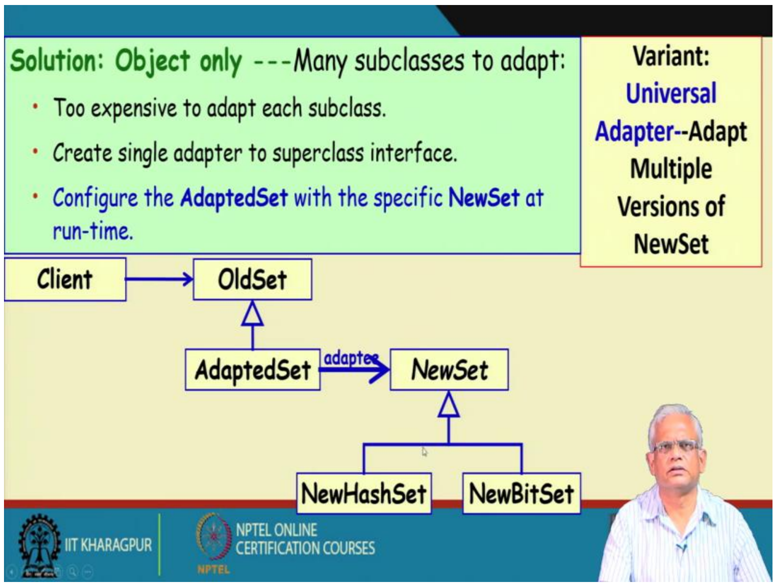
Outline of StudentComparator

```
public class StudentComparator
            implements Comparator<Student> {
    public int compare(Student s1, Student s2){
            return s1.score() – s2.score();}
    }
```

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The student comparator implements comparator and whenever there is a compare method, it finds the score of these two students and returns the difference of the score.

(Refer Slide Time: 08:10)



Solution: Object only ---Many subclasses to adapt:

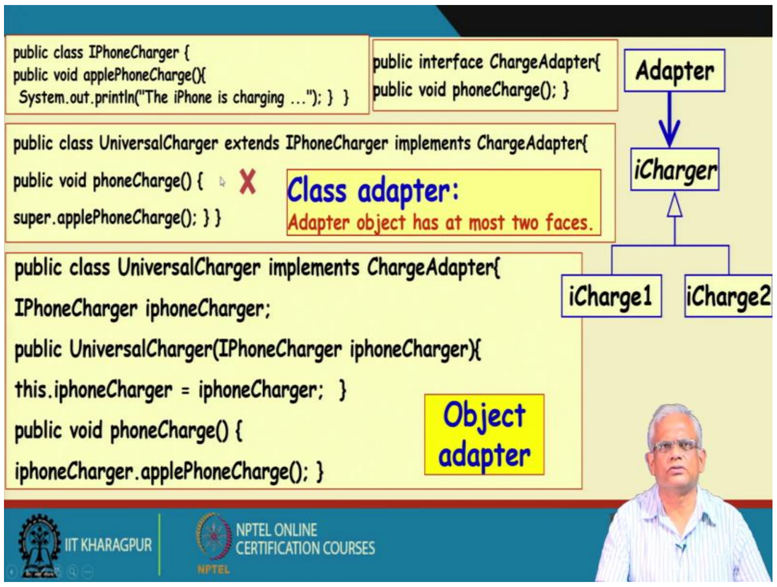
- Too expensive to adapt each subclass.
- Create single adapter to superclass interface.
- Configure the **AdaptedSet** with the specific **NewSet** at run-time.

Variant: Universal Adapter--Adapt Multiple Versions of NewSet

But what about universal adaptation? If a class needs to work with many different types of servers, each having different interface, then we need a universal adapter. Having many different types of adapters, we will be confusing for example, carrying a dozen adapters with you, as you go to different countries can be confusing picking out the required adapter and so on, we should have just one adapter and that is the universal adapter.

The main idea behind the universal adapter is that the adapter, the adapter here it implements the adaptee and it implements the target and has a reference to the adaptee and we have various types of adaptee here. So, it can work with any of these different adaptees. Let us see with some concrete examples.

(Refer Slide Time: 09:24)



```
public class IPhoneCharger {
public void applePhoneCharge(){
 System.out.println("The iPhone is charging ..."); } }
```

```
public interface ChargeAdapter{
public void phoneCharge(); }
```

```
public class UniversalCharger extends IPhoneCharger implements ChargeAdapter{
public void phoneCharge() {
super.applePhoneCharge(); } }
```

Class adapter:
Adapter object has at most two faces.

```
public class UniversalCharger implements ChargeAdapter{
IPhoneCharger iphoneCharger;
public UniversalCharger(IPhoneCharger iphoneCharger){
this.iphoneCharger = iphoneCharger;  }
public void phoneCharge() {
iphoneCharger.applePhoneCharge(); }
```

Object adapter

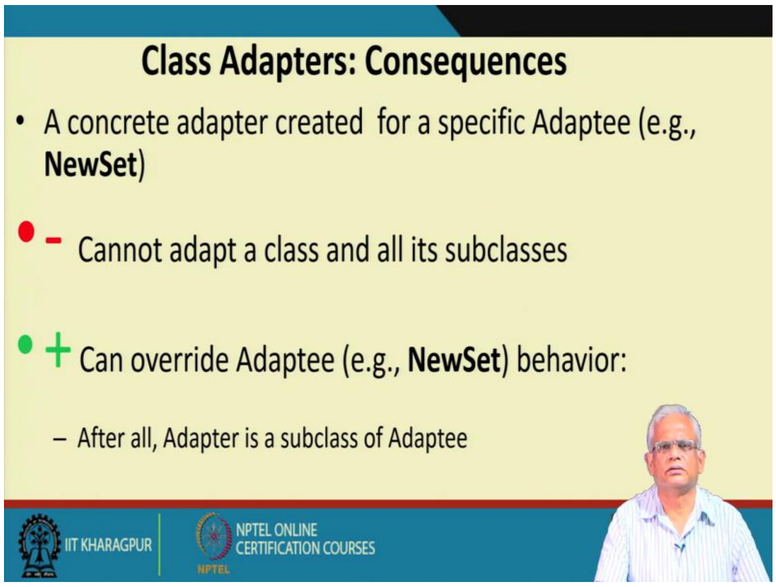
Let us say, we have a IPhoneCharger and here are the methods supported is Apple phone charger and then we have a ChargeAdapter, where we have the phone charge. In the universal adapter, if we have a UniversalCharger, then we would like to have it extend to the IPhone charger and then implements the charger adapter.

So, all the clients can call the PhoneCharge and then different types of chargers will be supported the IPhoneCharger, Android phone charger and so on. The client just calls phone charge and the universal charger should be able to call the corresponding charger. But then this is the class adapter and we know that it can only extend one type of adaptee and therefore it is a two phased it can only convert the target to only one of the adaptee. This cannot be used as a universal adapter.

The class adapter has at most two faces. It is not suitable to implement a universal adapter. Let us look at the object adapter. The object adapter implements the charge adapter and then it internally holds the reference to changer. In the constructor for the universal charger, it takes its argument to another charger and stores it internally. It may be a IPhone charger, it may be an Android charger and so on and then, once the phone charge is called, it just calls the corresponding method of the reference it internally holds.

So, the adapter just calls iCharger, the method on the iCharger and then there can be different type of charger like Android phone charger, IPhone charger, and so on. The class adapter is not suitable to implement a universal adapter and we need to use an object adapter.

(Refer Slide Time: 12:27)



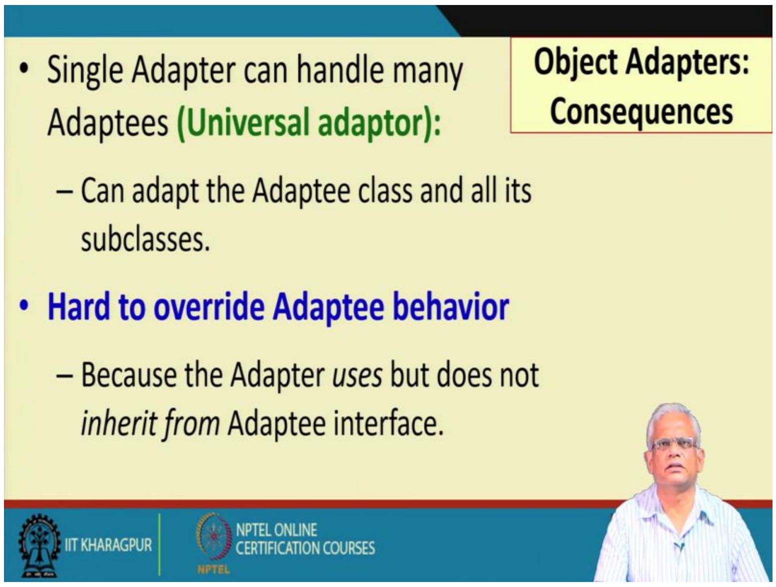
Class Adapters: Consequences

- A concrete adapter created for a specific Adaptee (e.g., **NewSet**)
- – Cannot adapt a class and all its subclasses
- + Can override Adaptee (e.g., **NewSet**) behavior:
  – After all, Adapter is a subclass of Adaptee

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The adapter makes two incompatible classes to work in the class adapter, a concrete adapter class is created. It cannot adapt a class and all its sub classes. That is, we cannot have a universal adapter realised by class adapters. But the advantage of the class adapter is that since the method is available internally, we can make some changes to the method, we can override the method, we did not call the same method, we can just override.

(Refer Slide Time: 13:16)



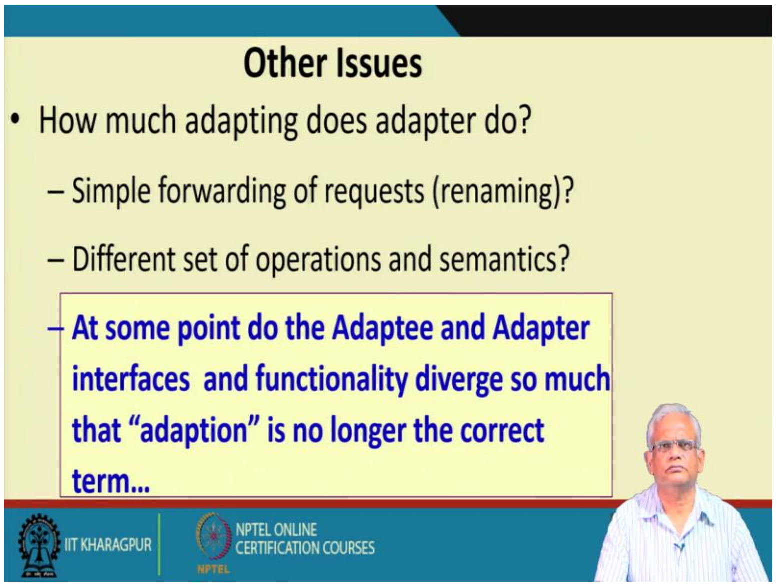
Object Adapters: Consequences

- Single Adapter can handle many Adaptees **(Universal adaptor):**
  – Can adapt the Adaptee class and all its subclasses.
- **Hard to override Adaptee behavior**
  – Because the Adapter *uses* but does not *inherit from* Adaptee interface.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

But the object adapter, the advantage is that we can realise universal adapter, but then it becomes hard to override the adaptive behaviour. Because we just want to call that method, we do not have the method available internally to make changes to that the data and so on, of the adaptee is not available internally and therefore to change the behaviour of the adaptee is difficult. If it is a straightforward calling the method then okay, the object adapter works fine. But then if we have to substantially change the behaviour, then the adaptee, then the object adapter cannot be used.

(Refer Slide Time: 14:14)



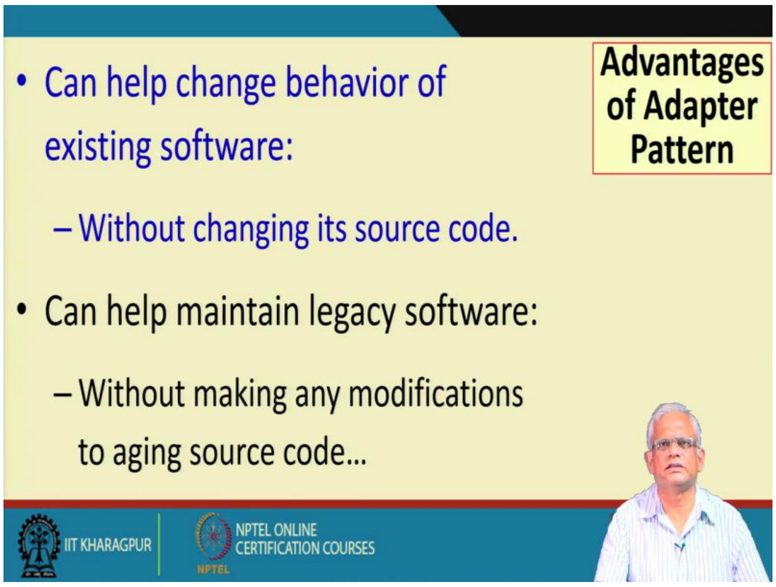
Other Issues

- How much adapting does adapter do?
  - Simple forwarding of requests (renaming)?
  - Different set of operations and semantics?
  - **At some point do the Adaptee and Adapter interfaces and functionality diverge so much that "adaption" is no longer the correct term…**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

We are just discussing about not just translating between the call from one object to another, but then modifying the behaviour of the call itself. One is simple forwarding the request or renaming and the other is having a modified operation. Typically, the adapter is not used to modify the operation because at some point of time, the functionality may diverge so much that adaptation is not the right word. To change the behaviour it is just not adapting the interface.

(Refer Slide Time: 15:02)



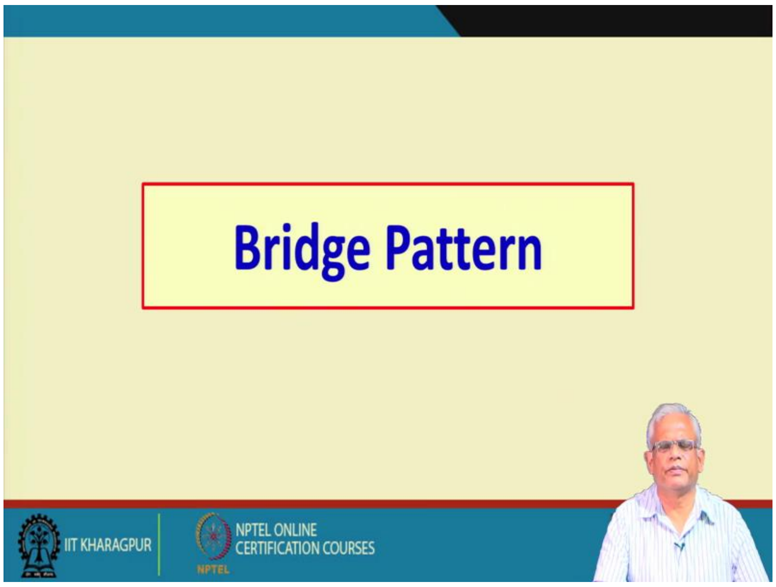
Advantages of Adapter Pattern

- Can help change behavior of existing software:
  - Without changing its source code.
- Can help maintain legacy software:
  - Without making any modifications to aging source code...

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

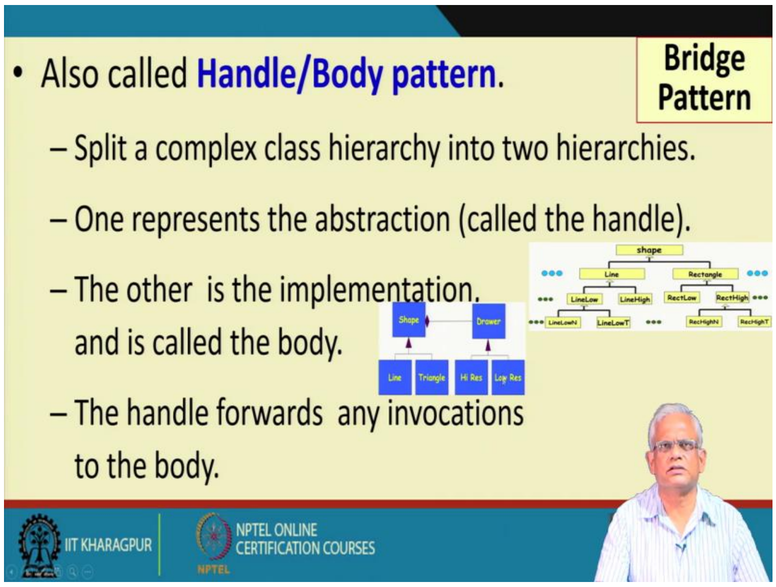
The object adapter, the adapter pattern makes two incompatible classes work without changing the source code of the existing client, very useful in maintaining legacy software without any changes to the ageing code. Because making changes to ageing code is a very difficult task. The structure would have already degraded that any change will have severe repercussions.

(Refer Slide Time: 15:38)



Bridge Pattern

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL



Bridge Pattern

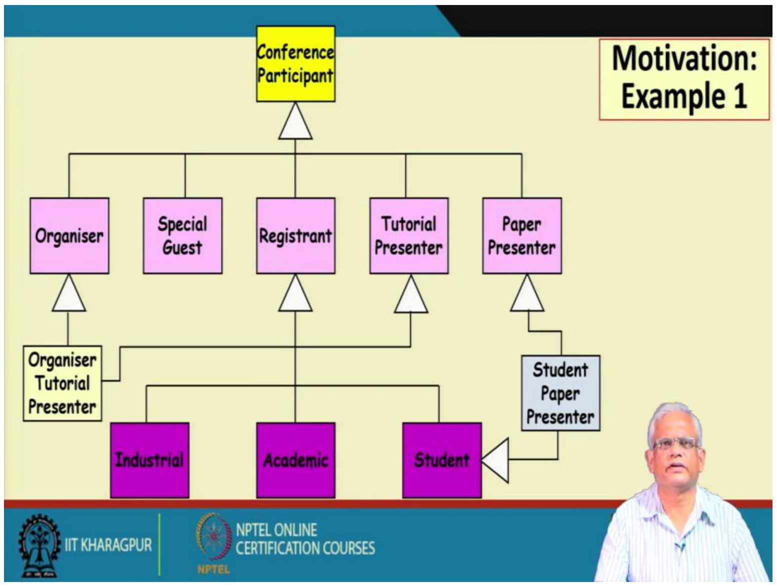
- Also called **Handle/Body pattern.**
  - Split a complex class hierarchy into two hierarchies.
  - One represents the abstraction (called the handle).
  - The other is the implementation, and is called the body.
  - The handle forwards any invocations to the body.

We have now discussed about the adapter pattern. Let us look at another important design pattern, which is the bridge pattern. The bridge pattern is also called as the handle body pattern. The main idea behind the bridge pattern is that, given a complex class hierarchy we split it into two simple hierarchies, one represents the abstraction, the abstraction hierarchy, which is called the handle and then the other hierarchy is called as the implementation hierarchy or the body and the handle hierarchy will forward any invocation by the client to the body hierarchy.

So, this is a complicated class structure, is a complex class hierarchy and using the bridge pattern, we will be able to split it into two simple hierarchies, which is much more manageable, very few classes observed here that there are 6 classes here compared to the large number of classes here, reduces the number of classes makes the design simple. We split a large complex hierarchy into two simple hierarchies, one we call as the abstraction hierarchy and the left side here and the other is the implementation hierarchy and then the abstraction hierarchy makes calls to the implementation hierarchy, whenever a client makes a call to the abstraction hierarchy.

It is a very useful pattern, if you use this pattern, the design can be simplified, easier to write code, easier to maintain and so on. The abstraction hierarchy here is called as the handle and the implementation hierarchy is called as the body of the pattern.

(Refer Slide Time: 18:14)

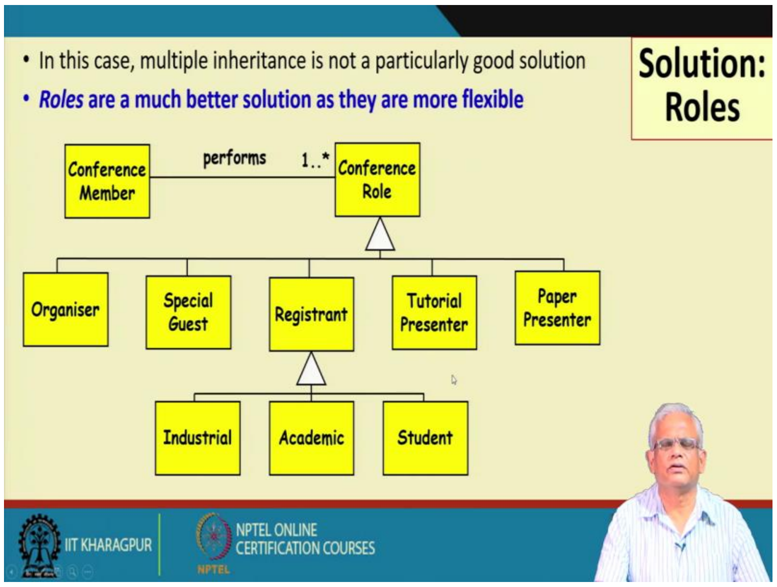


Let us look at an example to motivate the use of the pattern. Let us say we had written a software to manage the participants of a conference. In the class diagram for the conference automation, we had the conference participants classified into different sub classes. The organiser of the conference, the special invited guests, the registrant who register for the conference, the tutorial presenters, the paper presenters. But then, we found that the registrants can be 3 types, 3 main types, one is registrants from the industry, registrants from faculty of different academic institutions and also the students.

But then later we found that participant can be both a student and a paper presenter. A student also can present a paper and therefore, we had this class student paper presenter with basically features a both student and paper presenter. We also found that the organiser can also be permitted to present tutorials and therefore, we created this class organiser tutorial presenter, who is both a tutorial presenter and an organiser. Similarly, we can have an industrial paper presenter, industrial registrant and a paper presenter. We can have an industrial registrant and tutorial presenter and so on.

So, you can see that as we make the conference more flexible, the number of classes here increases drastically. This is a natural use of inheritance wherever we found that there is a special thing that has been permitted, we just extend this class hierarchy. But then the class structure becomes extremely complex, somebody trying to maintain the software will really find difficult to understand so many different classes.

(Refer Slide Time: 21:14)

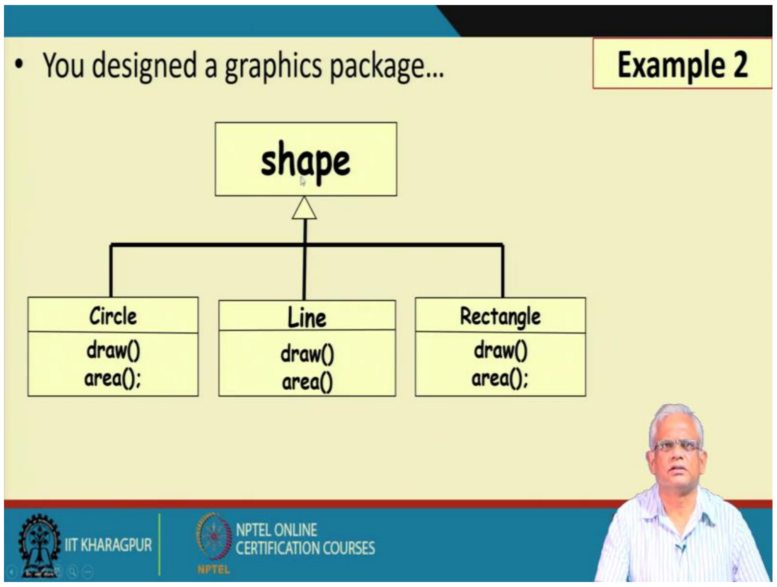


Solution: Roles

- In this case, multiple inheritance is not a particularly good solution
- *Roles* are a much better solution as they are more flexible

But using the bridge pattern, we can simplify the design we have the conference member has several rules, can be an organiser and a tutorial presenter, can be a registrant, academic registrant and paper presenter, a student and a paper presenter and so on. So, the conference member is one hierarchy and the conference role has another hierarchy. Now, let us see this pattern in more nitty gritty.

(Refer Slide Time: 21:56)

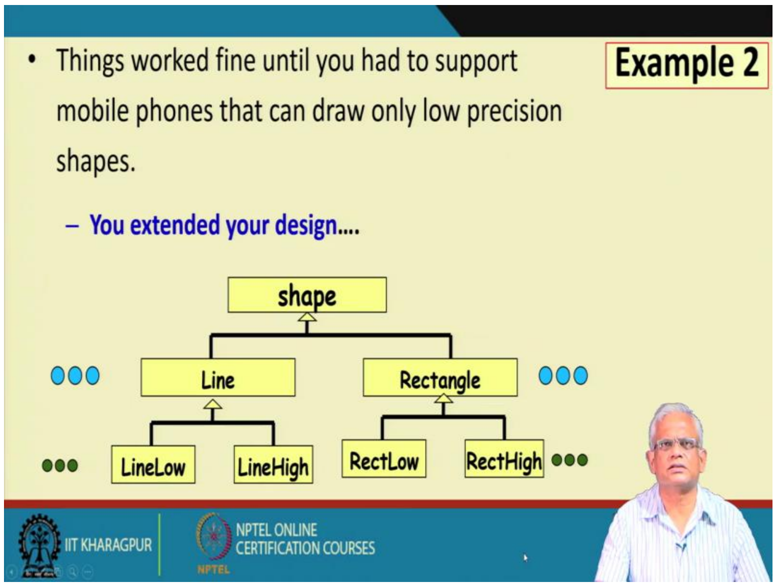


Example 2

- You designed a graphics package...

Let us say we have a drawing package and here are one of the basic class hierarchy is the shape class. The shape is an abstract class and there are many concrete classes like Circle, Line, Rectangle which support the methods specified by the Shape class.

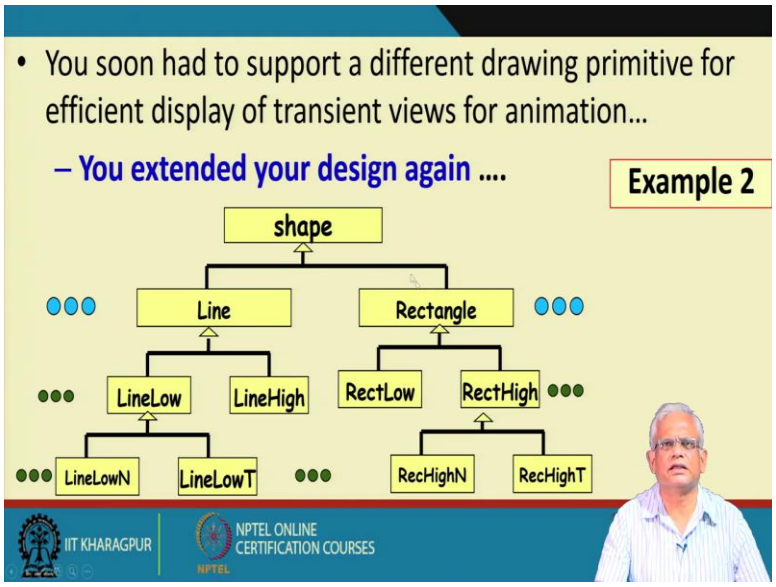
(Refer Slide Time: 22:30)



- Things worked fine until you had to support mobile phones that can draw only low precision shapes.
  - You extended your design....

But then, it was working fine until later, we found that we have to support mobile phones also for the drawing package and that mobile phone has a small screen and very low resolution. So, we had to maintain and said that see the line can be drawn on high resolution that is on desktops and on low resolution on mobile phones. Similarly, the rectangle we sub-classed into high resolution rectangle class and low resolution rectangle class and so on. So, by the maintenance to support mobile phone we have extended the hierarchy.

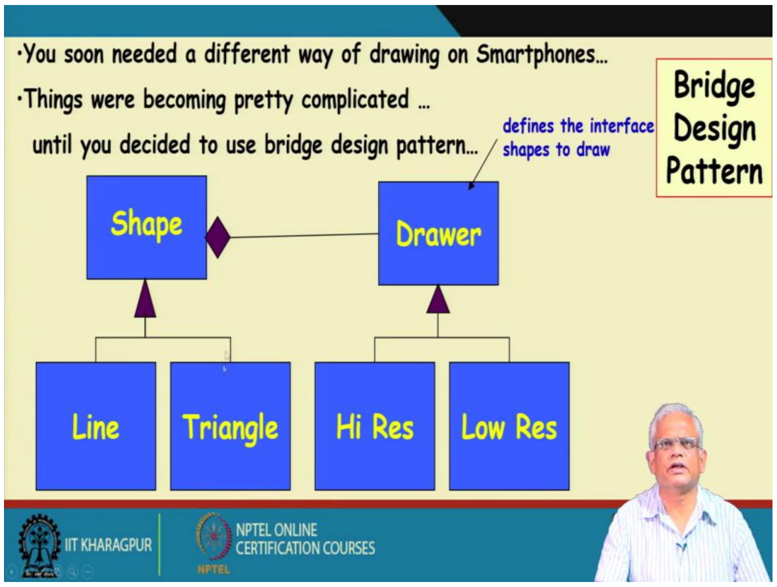
(Refer Slide Time: 23:29)



- You soon had to support a different drawing primitive for efficient display of transient views for animation...
  - You extended your design again ....

But then soon, we had another maintenance problem. We had to support different drawing primitives for efficient display of transient views for animation. Soon we had to support animation and we had to support transient view and therefore again we split, one is transient view and other is the permanent view and so on, we had to split every class here. The class hierarchy has become deeper and much more complex.

(Refer Slide Time: 24:13)



·You soon needed a different way of drawing on Smartphones...

·Things were becoming pretty complicated ...

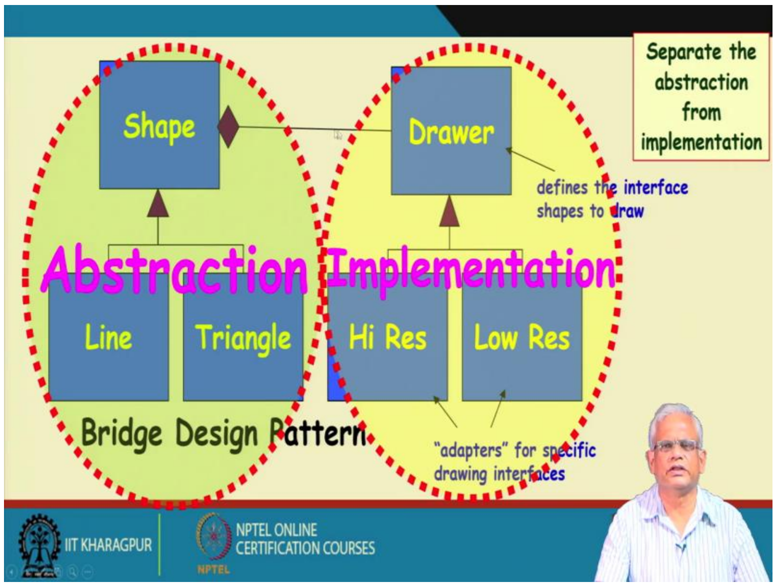
until you decided to use bridge design pattern...

Bridge Design Pattern
defines the interface shapes to draw
Shape
Drawer
Line
Triangle
Hi Res
Low Res
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Things were still working, but then we wanted to support smartphones. In the smartphone, we had a slightly different resolution requirement and we have to again split the classes. But then, the design had become extremely complicated. We had to redo the design and we use the bridge pattern.

In the bridge pattern, one is abstraction side, the shape is line triangle and so on. If we want to have more shapes added here. We will add it here on this side and the other is the implementation hierarchy, where the drawing, high resolution, low resolution, transient view, low resolution transient, high resolution transient and so on. Now, the number of classes has become very manageable and this is the bridge pattern.

(Refer Slide Time: 25:20)


Separate the abstraction from implementation
Shape
Drawer
defines the interface shapes to draw
Abstraction
Implementation
Line
Triangle
Hi Res
Low Res
Bridge Design Pattern
"adapters" for specific drawing interfaces
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The main idea of the bridge pattern is that it separates the abstraction from the implementation in a typical class hierarchy, the implementation and abstraction are there on one hierarchy and therefore, the number of classes uses almost exponentially as any maintenance problem comes in. For example, if we had to support an extra shape here, like a circle, then we had to create many classes in the complex class hierarchy when discussing, but here it just becomes one class there it may become 20 classes, just to support one different shape.

So, the bridge pattern, the main idea is that we split a complex hierarchy into two hierarchies. One is the abstraction hierarchy and the other is the implementation hierarchy and we have a bridge it appears like a bridge. See that line we have a bridge between the abstraction hierarchy and the implementation hierarchy by the appearance of this pattern, which is a bridge between abstraction and implementation hierarchy. It is called as the bridge pattern, a very useful pattern to simplify complex class hierarchies during maintenance.

Also, if we are aware of this pattern while writing our code, we will consciously try to separate out the abstraction hierarchy and the implementation hierarchy and have the bridge between them. We are almost at the end of this lecture. We will continue in the next class. Thank you.