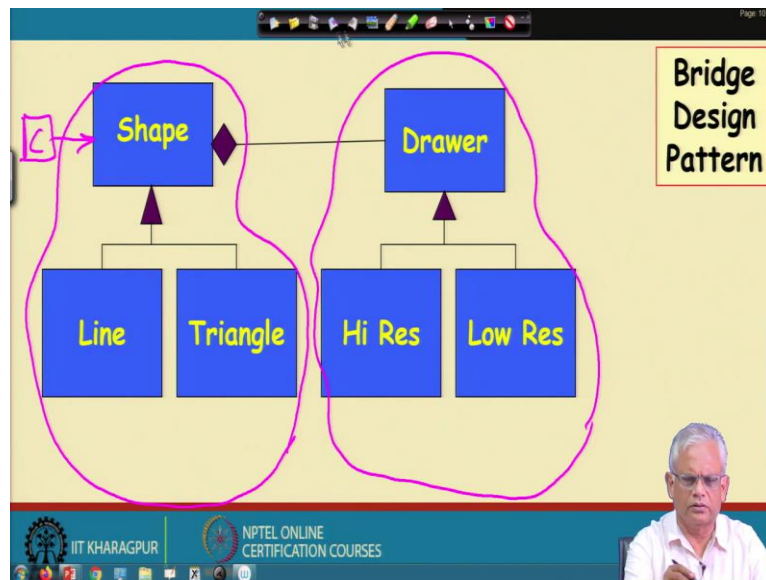
Object Oriented System Development using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 53
Bridge Pattern I

Welcome to this lecture! In the last lecture, we had just started to discuss about the Bridge Pattern. The bridge pattern is an extremely useful pattern, helps to simplify the design makes it more maintainable, extendable, very popular pattern. Let us look at this pattern. If you remember, we are discussing towards the end of the last lecture, the structure of this pattern with the help of some examples, let us just recollect the structure of the bridge pattern that we were discussing last time.

(Refer Slide Time: 01:00)



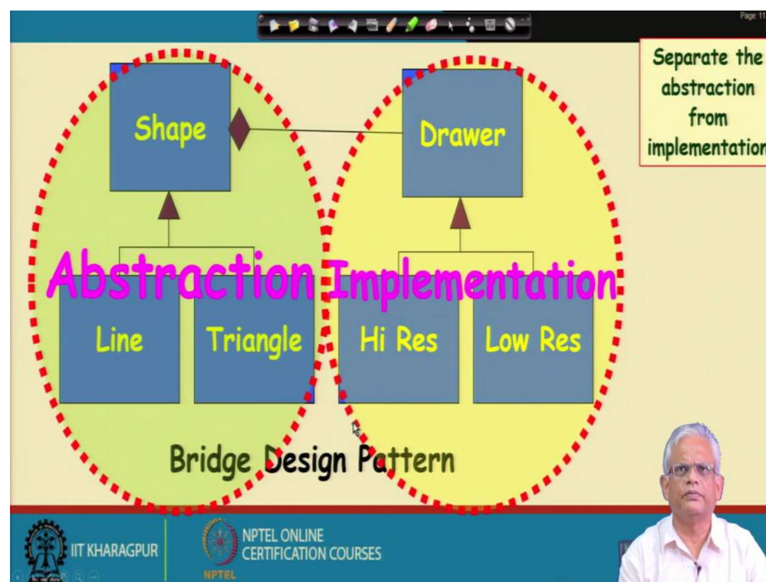
We had taken the example of a shape, various types of shapes; geometric shapes, line, triangle, circle and so on and then these are to be drawn on various platforms. On the desktops, mobile phones and so on and then we had discussed about the class hierarchy that kept on becoming more and more complex as we used different application like animation or we used a different platform and the solution to that we had drawn the bridge pattern.

Here, on the left side of the pattern, we have the abstraction hierarchy. These are the general concepts which typically do not change too much. On the right side, there is another hierarchy which is the implementation hierarchy and here, it is more likely to change as new platforms become available, new applications become available and so on and he had said that the number of classes here, if we use bridge pattern is quite manageable, understandable.

But if we do not use the bridge pattern, just use a class hierarchy as it is; then the number of classes grows very rapidly and becomes very complex. Here, the client class interfaces with the abstraction here, the abstraction hierarchy and then as the client invokes the methods and the abstraction, maybe to reshape, maybe to edit, move and so on.

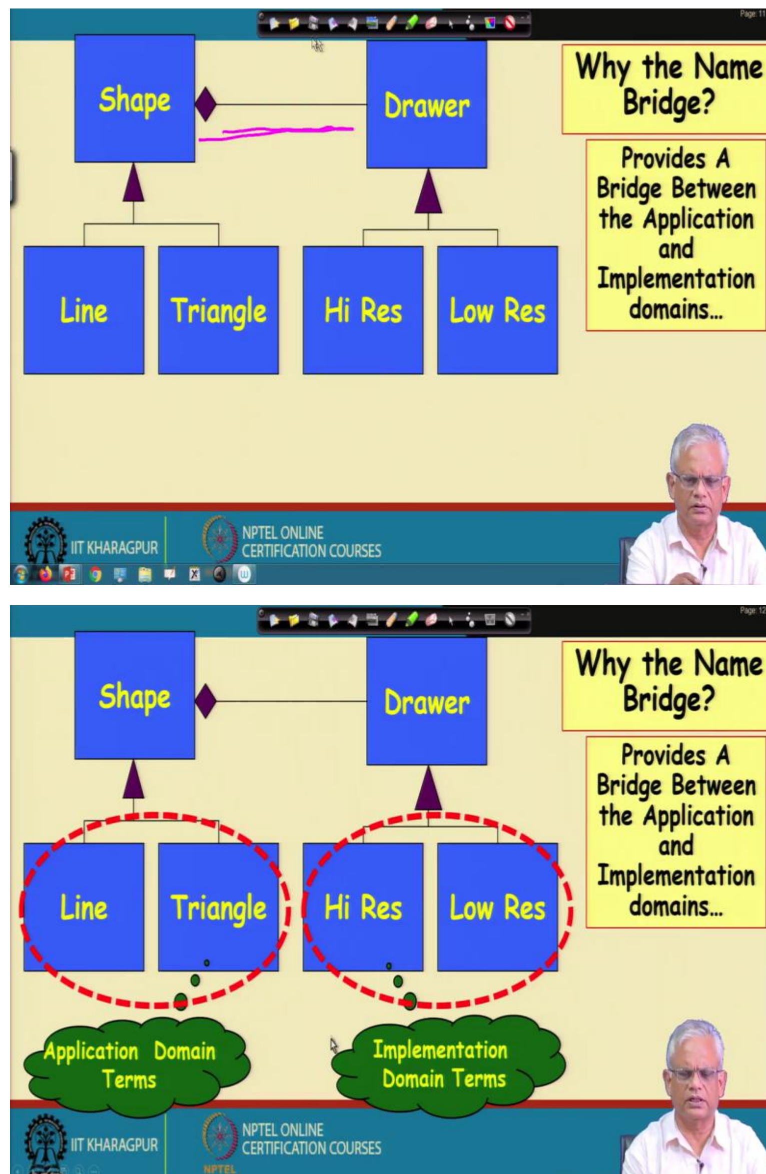
If required, the shape calls the corresponding concrete implementation class. We will see the details of this class in using more number of examples. But this was the main idea that we are trying to discuss in the last lecture.

(Refer Slide Time: 04:02)



So here, the main idea is that we do not use a single hierarchy where both there is a change to the abstraction or change to the implementation. These are handled uniformly the same hierarchy. Rather, in this pattern if we use the abstraction hierarchy is separate from the implementation hierarchy and any changes to the implementation hierarchy we just make that change without affecting the other hierarchy. So this side, the left side is abstraction hierarchy and the right side is the implementation hierarchy.

(Refer Slide Time: 04:53)

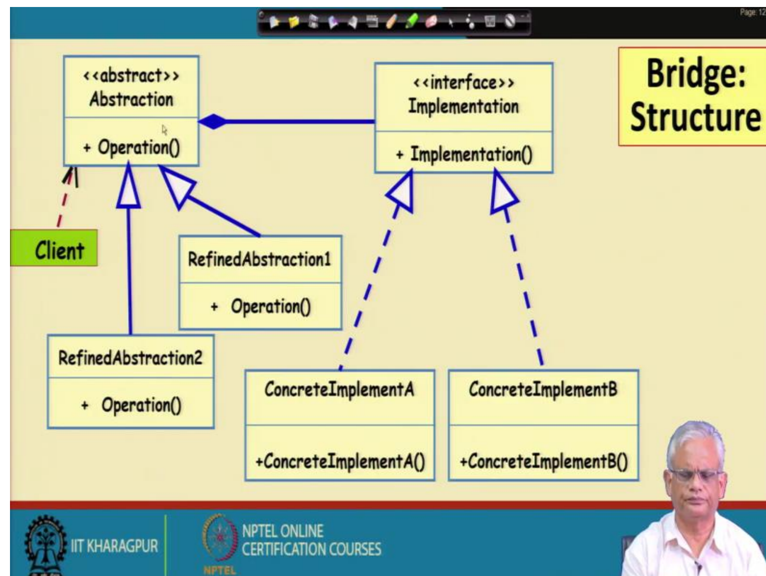


But then, why is the name 'bridge' given to this pattern? The name bridge is given because this pattern provides a bridge, a bridge between the abstraction hierarchy and the implementation hierarchy. So, this is the abstraction hierarchy where the terms line rectangle, these are invariant with respect to the specific implementation whether it is a desktop, whether it is a smartphone, whether it is an embedded application, whether it is an animation, we still call those as line and triangle because these are application domain terms.

On the other hand, on the implementation domain terms we have high resolution, low resolution, transient view and so on. So, this pattern separates out the abstraction hierarchy from the implementation hierarchy and provides a bridge between the abstraction and the

implementation which is basically an association or aggregation relation which helps the application class to invoke the methods of the implementation class.

(Refer Slide Time: 06:38)



We can redraw the structure like this, the same one. Here the abstraction is an abstract class and it is supported, many operations are defined, the refined abstractions are concrete class, these are application domain classes. The client invokes the abstract class here to which the corresponding concrete class is bound and then the implementation is an interface and depending on the type of platform and so on there are different implementation classes here and the correct implementation class should be bound to the abstraction.

(Refer Slide Time: 07:49)

Participants

- Abstraction
- RefinedAbstraction
- Implementor
- ConcreteImplementers

If we look at the participants of this pattern, we have the abstraction the refined abstraction which are basically concrete classes. In the application domain the implementation is an interface and then there are concrete implementations which are basically classes defined on platform variations and so on.

(Refer Slide Time: 08:26)

Participants contd...

- Abstraction
  - Defines the abstract interface
  - Maintains a reference to the implementer
- RefinedAbstraction
  - Extends the interface defined by Abstraction

The abstraction defines an abstract interface which the client uses and it maintains a reference to the implementer and that is how the bridge is established. The refined abstraction is a concrete class which extends the interface defined by the abstraction.

(Refer Slide Time: 08:52)

Participants contd...

- Implementer
  - Defines the interface for the implementation classes
- ConcreteImplementer
  - Implements the implementer interface

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

The implementation or the implementer this defines the interface of the implementation class and the concrete implementer are the concrete classes which are implementation specific and the implementer interfaces.

(Refer Slide Time: 09:13)

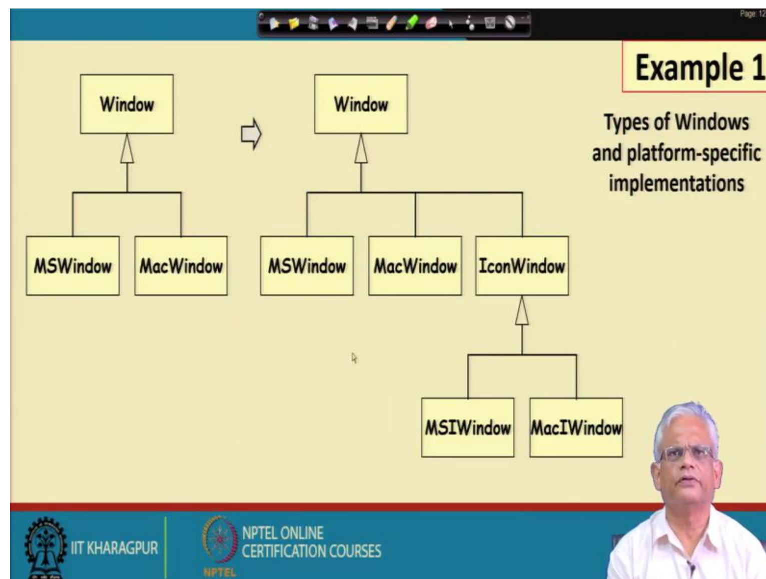
Collaborators

- Abstraction forwards client requests to Implementer object.
  - Clients interface with abstraction class.
  - Abstraction class forward any requests to the implementer class.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Here as the client invokes a method on the abstraction, it may do some work and also if necessary forward to the implementer object, the correct implementer object. The clients only interface with abstraction class, they do not interface with the implementation class and abstraction plus forwards in a request to the implementer class.

(Refer Slide Time: 09:48)

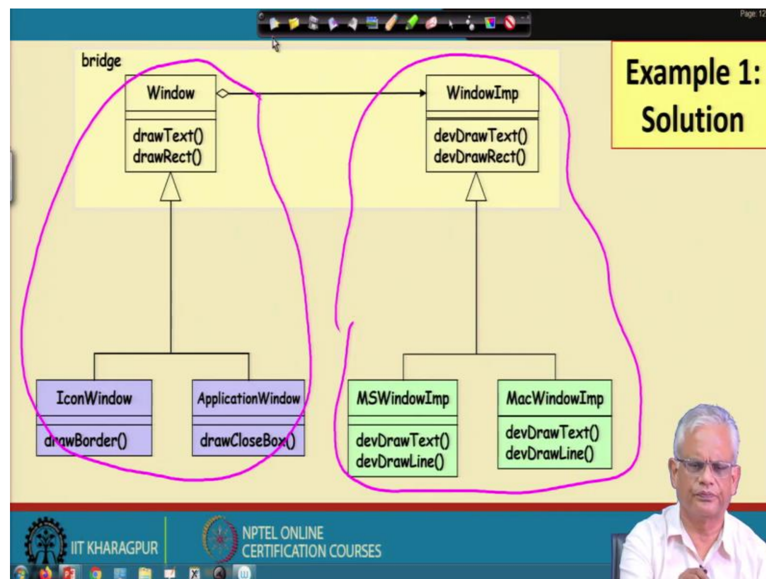


Let us take a few examples to understand the applicability of this pattern. First, let us look at Window example. In the Window example, let us say we had two window types, the Microsoft Window, Ms Window and the Mac Window but then, after some time we had the need for icons and we defined a special type of window called this icon window.

But then the icon window must be implemented on both the MS Window and the Mac Window and therefore, we sub-classed the Mac Icon Window, made it MSI Icon Window and the Mac Icon Window. And similarly, we might have other platforms for example, Android platform and then we not only add class here Android Window but also we will have to add the Android Window here. Similarly, Unix Window will also have to add here. So, each time we make a change, multiple class additions are required. There is a single class hierarchy and we have not separated the abstraction from the implementation. That is the problem, we have not use the bridge pattern, to be able to use the bridge pattern we must identify what is the abstraction here and what is the implementation.

The abstraction are the different types of windows, for example the icon window, the application window, these are the different types of windows and their implementation whether it is in the Microsoft platform Mac or maybe the Unix platform, Android and so on those are the implementation. If we consider that then the application of the bridge pattern becomes very simple.

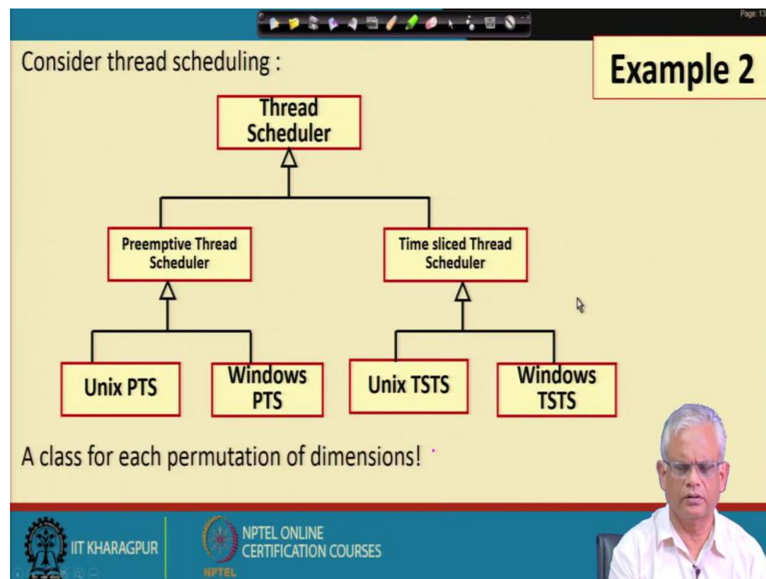
(Refer Slide Time: 12:18)



We have the abstraction on the left side there are two main types of window concepts. One is application window and one is the icon window. Each icon has its own window and the application window has many icons and other things. So, this is one type of window and this another, these are two basic types of windows, but then this can have implementations and various platforms.

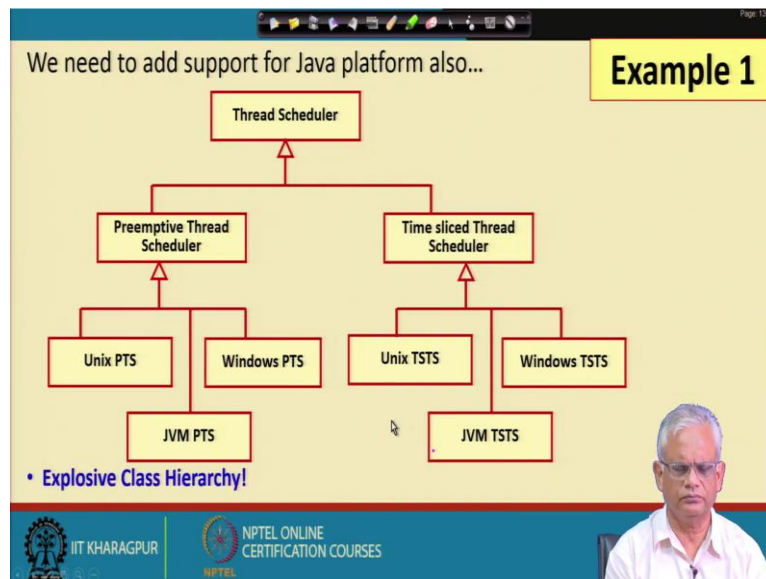
So, we have the window implementation that is interface and we have various implementations of the window implementation that is Microsoft window implementation, Mac window implementation, Android window implementation, Unix window implementation and so on. So, we have separated out the abstraction hierarchy from the implementation hierarchy and have successfully applied the bridge pattern.

(Refer Slide Time: 13:39)



Now, let us look at another example. We are given these hierarchies which some designer has come up. So, he has the thread scheduler class here and then he sub-classed it into preemptive scheduler and time sliced scheduler and then there is a Unix preemptive thread scheduler, the windows preemptive thread scheduler. There are two types of preemptive thread scheduler; one for the Unix platform and another for the window platform. Similarly, for the time sliced thread scheduler will have two sub-classes; the Unix time sliced thread scheduler and the windows time sliced thread scheduler. But later as maintenance was required, it was used for some time and then new requirements arose.

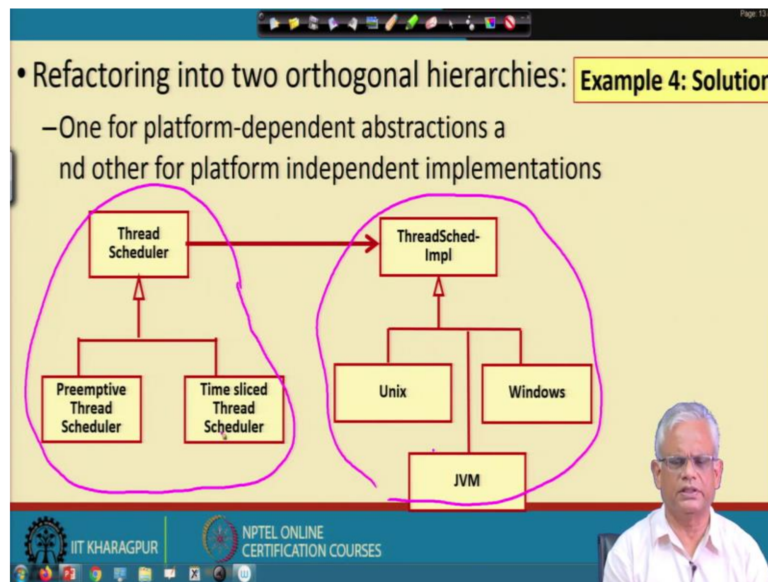
(Refer Slide Time: 14:51)



And then we need to support the Java platform and once we try to support the Java platform, we saw that we not only should have a Java JVM preemptive thread scheduler, but also the JVM time sliced thread scheduler. So, the hierarchy becomes more and more complex as new changes occur to either the abstraction or the implementation concepts, we need to apply the bridge pattern, separate out the abstraction and the implementation and provide a bridge between that.

Now, what is the abstraction here? The abstraction are the different types of thread schedulers, it is very easy to see here there is simple example that there are two basic types of thread scheduler; one is the preemptive scheduler and the other is the time sliced scheduler. But then their implementation is different on different platforms. For example, Unix windows JVM. So, these are the implementation domain and this is the abstraction domain.

(Refer Slide Time: 16:18)



If we separate that out, we have the abstraction hierarchy on the left side and the implementation hierarchy on the right side. The number of classes has reduced and not only that, if there is a change on the abstraction side, we have a new scheduler or we have a new platform, we just make change, small changes to the corresponding class hierarchy, we do not change; need to have too many sub-classes added for just one simple change. Now, let us look at one more example.

(Refer Slide Time: 17:07)

- Suppose an abstraction has several implementations:
  - Inheritance is commonly used to accommodate these!!!

Observation

1. Inheritance binds an implementation to the abstraction permanently:
   - It becomes difficult to modify and reuse abstraction and implementations independently.
2. Inheritance without a **Bridge**:
   - Leads to violation of single responsibility principle (SRP)

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

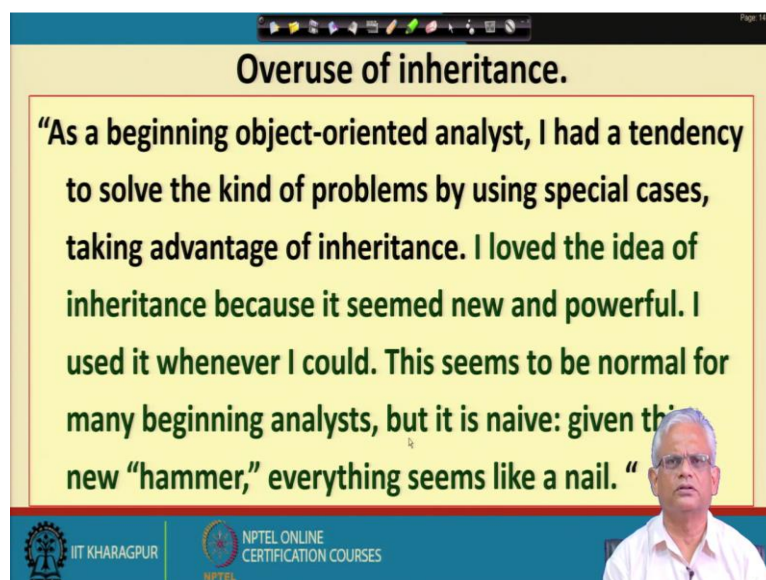
Before that from our examples, let us make some observations. If an abstraction has several implementations that is we have defined some abstraction ideas for application, but then it

can be used on various platforms and so on and then the programmer is typically tempted to use inheritance to solve the problem, just add some classes in the inheritance hierarchy.

But then, the problem that occurs is that the inheritance binds the implementation and abstracts together permanently and it becomes very difficult; the solutions are too complex, any small changes require too many changes. But then, what is the design principles does this kind of solution violet that a programmer to for any small change that is on the abstraction hierarchy or the implementation hierarchy, just make sub-classes on the same class hierarchy.

If you look back to the previous lectures, you will see that the, it violates the single responsibility principle, violates the single responsibility principle.

(Refer Slide Time: 18:53)



Now let us see what a programmer had to tell as he became more experienced and observed. What he has been doing, he had this to say, he said, as a beginning object oriented analyst I had a tendency to solve the kind of problems by using special cases, taking advantage of inheritance I loved the idea of inheritance because it seemed simple, new and powerful. I used it whenever I could. This seems to be normal for many beginning analysts. But it is naïve given this new hammer, this new tool somebody has learnt very simple tool everything seemed like a nail. Very correct and that is how many of the application solutions become complex because of the overuse of inheritance tend to violate the single responsibility principle and must keep in mind the bridge pattern and make use of that and given an ugly design, we should be able to refactor by using the bridge pattern.

(Refer Slide Time: 20:17)

Bridge: Applicability

Use bridge Pattern when:

- You want to avoid a permanent binding between an abstraction and its implementation.
  - Implementation may be selected or switched at run time.
- Both the abstraction and their implementation should be extensible by subclassing without impacting the clients:
  - Even code should not be recompiled.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

We must use the bridge pattern when we want to avoid a permanent binding between abstraction and implementation. Also, the implementation may be selected and switched at runtime. So, the application can run on different platforms seamlessly. Both the abstraction and their implementation are extensible without implementing the clients, even the code did need not be recompiled.

(Refer Slide Time: 20:54)

Bridge Pattern: Example 3

- Circle Shape problem
  - Different implementations for drawing circle
  - A method for changing the circle abstractly
- Participants
  - Abstraction
    - Interface Shape
  - RefinedAbstraction
    - Class CircleShape
  - Implementation
    - Interface DrawingAPI
  - ConcreteImplementations
    - Class DrawingAPI1
    - Class DrawingAPI2

<<abstract>> Abstraction
+ Operation()
<<interface>> Implementor
+ Implementation()
RefinedAbstraction
+ Operation()
ConcreteImplementA
+ Implementation()
ConcreteImplementB
+ Implementation()

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

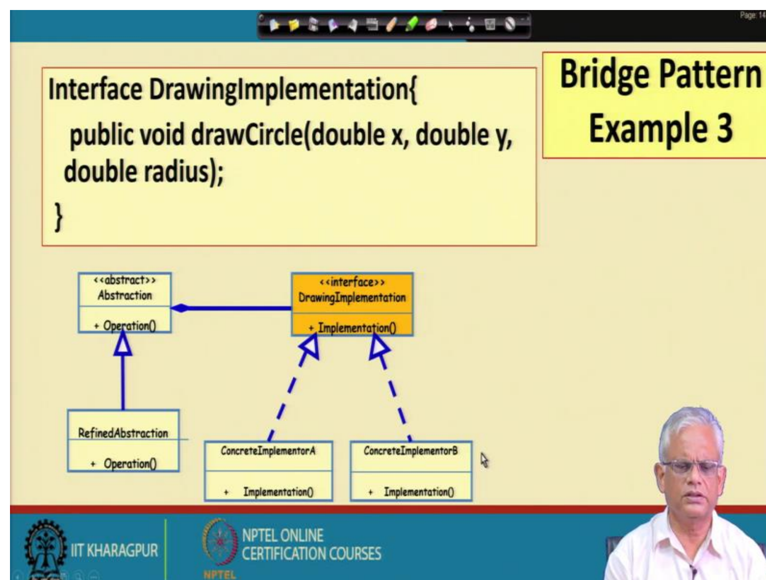
Now let us look at another example. Let us look at the circle shape problem. The problem is that we have a circle; we have a package where we can draw a circle and also we can do various manipulations on the circle for example, we may stretch it, we may contract it, we may change the colour, we may change the fill pattern and so on. But also the circle can run

on the browser. The circle can be displayed on the browser on a standalone application on the mobile phone, and so on.

You say, again straightforward application of the bridge pattern. So, here the abstraction and the refined abstraction is the shape, circle shape and the concrete sun circle shape where we can have various operations like resize, change fill pattern, change colour and so on and then the implementation hierarchy where we have implementation on web browser, smartphone and so on.

So, here, these are the classes which participate in this pattern, the abstraction which is the interface, refined abstraction which is the concrete circle class. Then we have the implementation which is the interface and then the concrete implementations which use different drawing APIs for different platforms.

(Refer Slide Time: 22:52)



Now, let us try to see the code, the Java code. Here on the implementation side, the drawing implementation is interface; it only supports draw circle given the centre and the radius x, y and the radius it can draw it on a web browser that is one implementation another is on smartphone, another is on specific application and so on. So, this is the simple code for the interface drawing implementation.

(Refer Slide Time: 23:41)

Bridge Pattern Example 3

```
class ConcreteImplementerA implements DrawingImplementer{
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

class DrawingAPI2 implements DrawingImplement {
    public void drawCircle(double x, double y, double radius){
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
```

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

And then we have the concrete implementations which implement the interface drawing implementer. So, we have the concrete implementer A, implements the drawing implementer and here for each of the methods in the interface we provide the definition and similarly for the other concrete implementation, we have the implementation for the draw circle.

(Refer Slide Time: 24:14)

Bridge Pattern: Example 3

```
interface Shape {
    public void draw();
    public void resizeByPercentage(double pct);
}
```

<<abstract>> Shape
+ Operation()
<<interface>> DrawingImplementor
+ Implementation()
RefinedAbstraction
+ Operation()
ConcreteImplementorA
+ Implementation()
ConcreteImplementorB
+ Implementation()

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now let us look at the abstraction hierarchy. Here we have the interface shape. Here we have the abstraction domain terms or the application domain terms like draw, resize, change fill colour, fill pattern and so on. Now, this is a simple interface shape.

(Refer Slide Time: 24:48)

Bridge Pattern Example 3

```
class CircleShape implements Shape { /** "Refined Abstraction" */
    private double x, y, radius;
    private DrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius, DrawingAPI
    drawingAPI)
    { this.x = x; this.y = y; this.radius = radius; this.drawingAPI = drawingAPI;
    }
}
```

Bridge Pattern Example 3

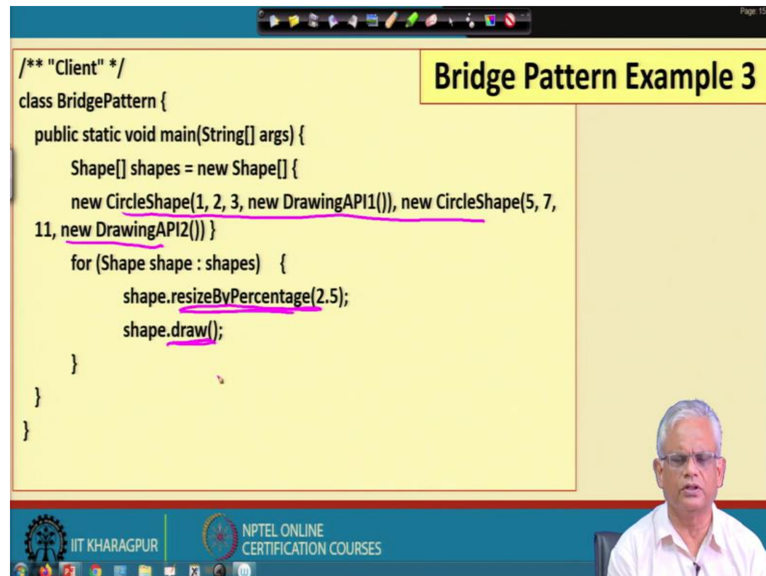
```
class CircleShape implements Shape { /** "Refined Abstraction" */
    private double x, y, radius;
    private DrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius, DrawingAPI
    drawingAPI)
    { this.x = x; this.y = y; this.radius = radius; this.drawingAPI = drawingAPI;
    }
    // low-level i.e. Implementation specific
    public void draw() {
    drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct)
     radius *= pct;
     }
}
```

Now let us look at the concrete shape which is the circle shape, circle shape implement shape and then it has constructor where we provide the specific concrete implementation. The drawing API is an argument on the constructor and that is how the bridge is established and here in the private drawing API, we store the argument provided in the constructor and then we have the other application specific, abstraction specific methods.

Wherever required for implementation specific methods, we call the method of the class, specific class and the implementation hierarchy. Here we are calling the drawing API.drawCircle which is the drawing API is the implementation class plus concrete implementation class and we are calling the draw circle on that.

Whereas the other methods are on high level abstraction specific methods. For example, resize by percentage, change fill pattern, change circle colour and so on. Simple code example for the bridge pattern.

(Refer Slide Time: 26:43)



Now, this is the client code. In an array we create many circles, two circles we have created and we have given two different APIs for those two circles and the client just interfaces with the abstraction domain class which is the shape and then we call the shape resize by percentage, it will be handled by the method in the abstraction class without really invoking a method on the implementation class.

Whereas we might also invoke, the client might also invoke some methods which will require calling a corresponding method on the implementation hierarchy. We are almost at the end of this lecture we will just stop here. We will look at some broad conclusions on the bridge pattern and then we look at a different pattern. We stop here. Thank you.