

Object - Oriented System Development using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 44
Observer Pattern 2

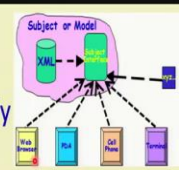
Welcome to this session! In the last session, we had started to discuss about the Observer Pattern. The Observer pattern is an important pattern and has many applications. We had discussed to some extent, let's quickly revisit what we had discussed and we will proceed from there.


(Refer Slide Time: 00:43)


- **Problem:**
 - When a model object changes state asynchronously and is accessed by several view objects:
 - How should the interactions between the model and the view objects be structured?


- **Solution:** Define a one-to-many dependency, so that when model changes state, all of its dependents are notified and updated automatically.

Observer Pattern



IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES



The problem that the Observer pattern tries to address is that when the model object changes state asynchronously and is accessed by several view objects, how should the interactions between the model and the view objects be structured. Just to tell little bit about the problem, a good solution for any software consists of a layered solution, at the bottom of the layer is the model, the model or the subject is the one which keeps track of the data, any update on the data is updated on the model and there are several view objects.

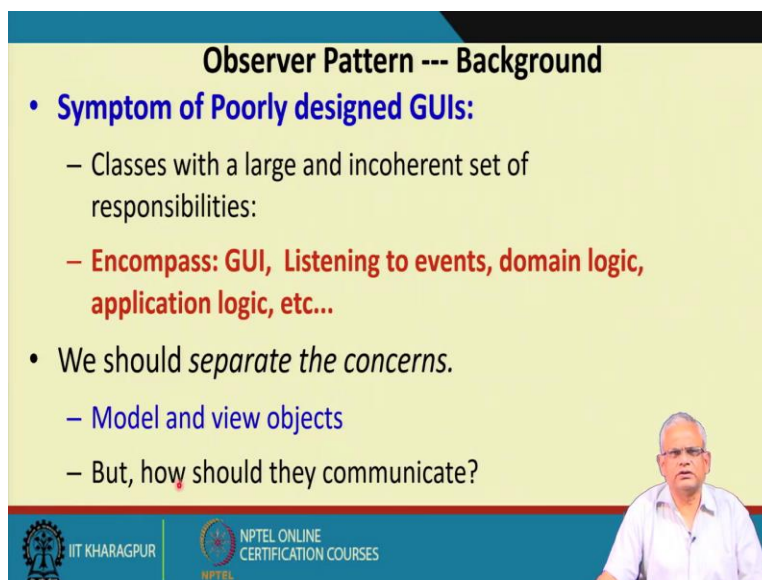
The view objects may be of different types as in this example and when the model object changes. For example, we have a database keeping track of various student records and then we have various applications running here. One is possibly grid computation, one is fee computation, another is awarding prizes, keeping track of attendance and so on.

Typically, we have a layered solution to problems like this and we have the model objects which keep track of the data and the view objects. In a typical situation like the one we mentioned the one we mentioned to put the student database. The View objects they pull the data from the model object, any necessary data they pull here by sending a request and do whatever they need to do display and so on. But then the problem here is that the model object state changes asynchronously in the student example, the student state does not change asynchronously, the grade of the students get reflected at the end of the semester they do not every now and then.

So, the pull solution works will for example select student database etc. But then there are many applications as we will see that the model keeps on changing and we do not know when that will change and the view has to have a consistent display of the model whenever the model gets updated the view also should also get updated and in these situations, how should the communication be structured i.e., the problem that the Observer pattern addresses.

The solution that is defined by this pattern is that there is a one-to-many dependency between the model object and view object. The different view objects are dependent on the model object. So, that is many-to-one dependency and when the model changes, it updates all the views simultaneously the model knows when there is a change occurs and it updates the views in contrast to the previous solution we said, where the view objects pull the data from the model.

(Refer Slide Time: 04:52)




Observer Pattern --- Background

- **Symptom of Poorly designed GUIs:**
 - Classes with a large and incoherent set of responsibilities:
 - **Encompass: GUI, Listening to events, domain logic, application logic, etc...**
- We should *separate the concerns*.
 - Model and view objects
 - But, **how** should they communicate?

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

NPTEL

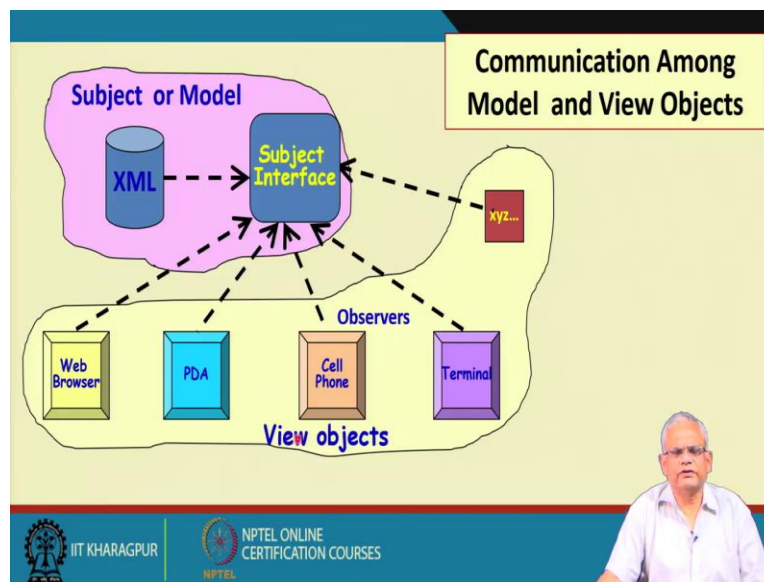


The Observer pattern helps to develop a good design of an application. If we are not careful, we might get a very poorly design GUI where the Observers or the GUI, they not only do the functions of GUI, but also listen to events they have domain logic built into that, application logic etc and this is an example of bad code because it will be very difficult to maintain this code, understand and maintain this code.

We need to separate out or layer a distinct layer should be there between the model and the view. The view should be concerned with only the user interaction that is displaying to the user that is the view and the model is the one which keeps track of the data. But this Observer pattern tries to address how the model and view should communicate.

The model view layering is not new. It is a well-accepted design solution, layering between the view and the model, the views should be concerned with only display. The model should be only concerned with how to keep the data. The problem that this Observer pattern addresses is that how should the view and the model objects communication?

(Refer Slide Time: 06:41)



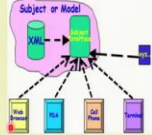
This is a typical layered solution where we have the subject or the model keeps track of the various data, maybe in XML database or maybe in a DMS or in a file and the data changes asynchronously. Maybe there are mouse clicks, maybe there is a communication line and so on, the data changes. But the problem that this pattern tries to address is that how do these view objects, they communicate with the model objects. The pull from above will not work because


the view objects would not know when exactly changes to the model occur, they occur asynchronously.



(Refer Slide Time: 07:36)

**Model View
Separation Patterns:
Solution Overview**

- Model objects should not invoke services of view objects.
- Vice-versa is OK
- View objects are transient
- Reuse, extension, maintenance are frequent.
- A change to a view object should not require change to the model object.








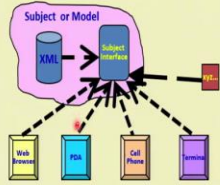
So, here the solution is that, the solution proposed by the Observer pattern is that, if we have the model object hard coding the ideas of different view objects and updating them is not a good idea. That is a poor solution. The pull from above is an okay solution, but does not work for asynchronous update of the data.

For asynchronous update of the data, if we reverse these arrows and have the model object keep track of the ideas of the view objects is not okay. Why is that the reason is that, if we hard coat the references of the view objects then it will be very difficult to change, we cannot have new types of views attached, some views dropping out and so on.

In a static scenario, where we have only a fixed set of Observers and they do not change, there is no chance of change to the view Observer overtime. Then possibly a push solution might work that the model hard codes or keeps track of the ideas of the view objects and calls the view objects when there is a change, but we are considering the situation which is dynamic view objects may switch off or more view objects may get added. There are many applications which require this kind of situation as we will see.

(Refer Slide Time: 09:29)

- **Most obvious solution:** **Solution 1: Pull from Above**
 - View (boundary) objects invoke model objects.
- **Problem:**
 - Views can become inconsistent
 - View object does not know when model state changes




IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The pull from above as we said works only in asynchronous update case that the view objects know when exactly the update takes place and they can pull the data. But in a synchronous case, the view objects do not know when the updates will take place and therefore, they cannot just pull from above.

(Refer Slide Time: 09:57)

Context in Which Pull from Above Does Not Work

- **Data changes asynchronously:**
 - An event in a simulation experiment, stock market alert, network monitor, etc.
 - A mouse event causes change to a shape, etc.
- **Dynamic set of observers**



IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

There are many situations where the data changes asynchronously. For example, in a simulation experiment we do not know when exactly a simulation activity will get over and the data in the

simulation engine will get updated. In a stock market we do not know when exactly stock trading will take place at what price.


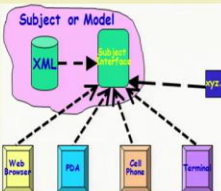
In a network monitor, you do not know when intrusion will occur, occurs asynchronously no idea when it will occur or if you are using a graphics editor, maybe the user will give input by a keyboard, we do not know when the input will come by keyboard or by a mouse click, we cannot predict when changes will occur, but then the view should be updated and the views can be created dynamically; one view maybe a 2D, another maybe a 3D view, another maybe a view from a different perspective and so on. We do not know how many view objects will be there.

So, that is a dynamic set of Observers. So, the Observer pattern needs to be applied when these two conditions are satisfied. We want to solve the communication problem between the view and the model objects and the model objects change asynchronously and the view objects are dynamic the sense that new view objects may appear some view objects may drop out and so on.

(Refer Slide Time: 11:52)

Observer Pattern: Context

- There could be many observers
 - Also the number of observers may vary dynamically...
 - **Each observer may react differently to the same notification**
- Subject (model) should be as much decoupled from the observers as possible:
 - **Allow observers to change independently of the subject**

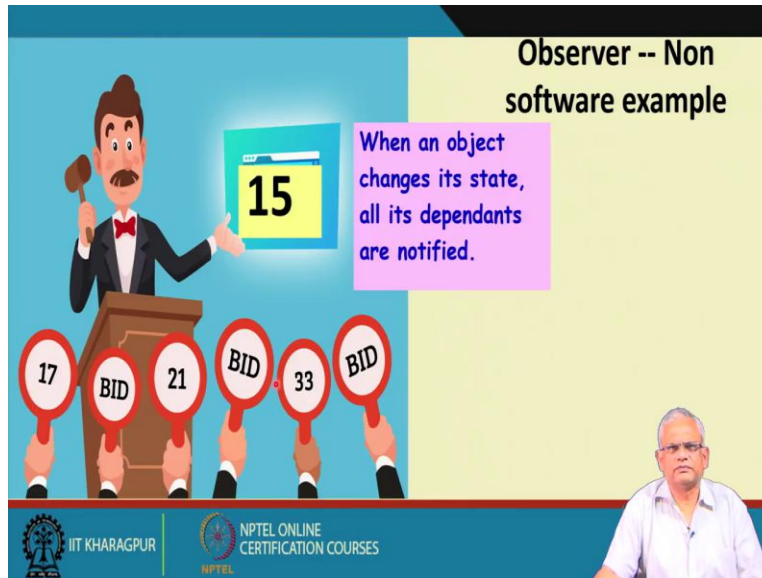


NPTEL ONLINE CERTIFICATION COURSES

And also, the view objects may react differently to the same notification not only they vary, but then the model cannot really take care of what activity will take place in the view objects. Because they may vary across the view objects one may be a simple mobile phone another may be a sophisticated smartphone, large screen another maybe a desktop, another maybe a laptop and so on and in response to a change to the model object, the action that takes place in the view objects or the Observers may be different.

The solution that the Observer pattern proposes is that we have to decouple the view from the model, we cannot hard-code the references of the view objects in the model object because that will eliminate the flexibility that we can add more Observers or take away some of the Observers. We need to have a solution where we can add more Observers different types of Observers, the Observers may change, the subject may change and so on.

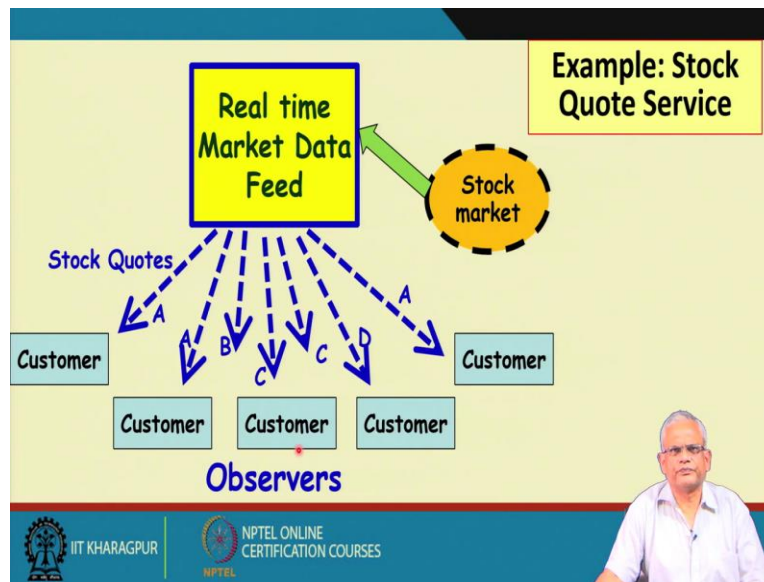
(Refer Slide Time: 13:26)



An analogy of this situation is an auction, where there are a set of people bidding and the person conducting the auction. The persons who are bidding, they cannot see who is bidding at what price but the person conducting the bid can see that updates the latest price and based on that further bidding takes place, finds out what is the highest at that moment 33 and then updates 33 here and as he updates all the viewers can see that.

The number of bidders can vary and we do not know when the changes will take place. The Observer pattern works in a similar way that the number of viewers can vary the changes to the model can occur asynchronously and when it occurs all the persons bidding have to be notified.

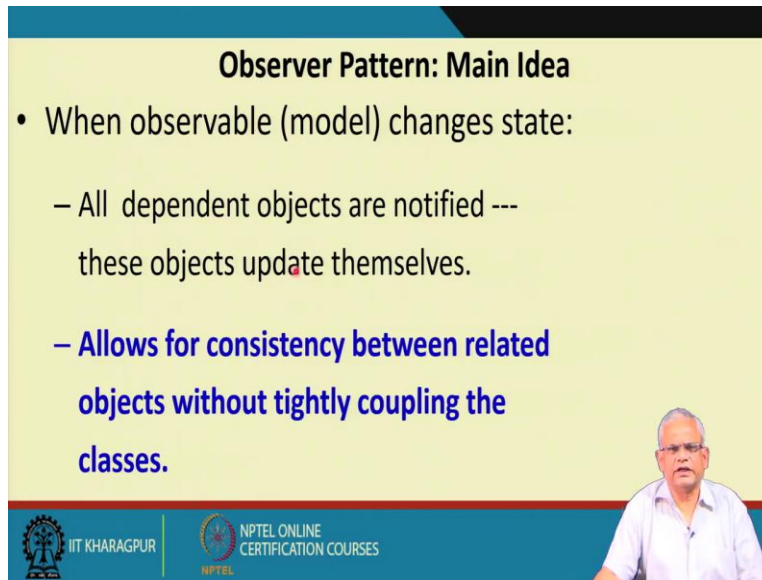
(Refer Slide Time: 14:43)



Another example is the stock market. Lot of stock trading takes place in the stock market and these are requested by various Observers. There is a real-time data feed from the stock market. But then the different Observers or the customers are interested in different stock. Some are interested in stock A, some are stock B, some are in stock C, D etc. Now the problem here is that when there is a change, a new price is there in the stock market.

Somehow the computer here has to notify the corresponding users about a change in the price of their stock and also the number of customers can vary because they are using a web browser to observe more customers can switch on and start observing the stock market. So, this is another situation, where the data changes asynchronously when trading takes place is not predictable and also how many customers will be there is not predictable.

(Refer Slide Time: 16:09)



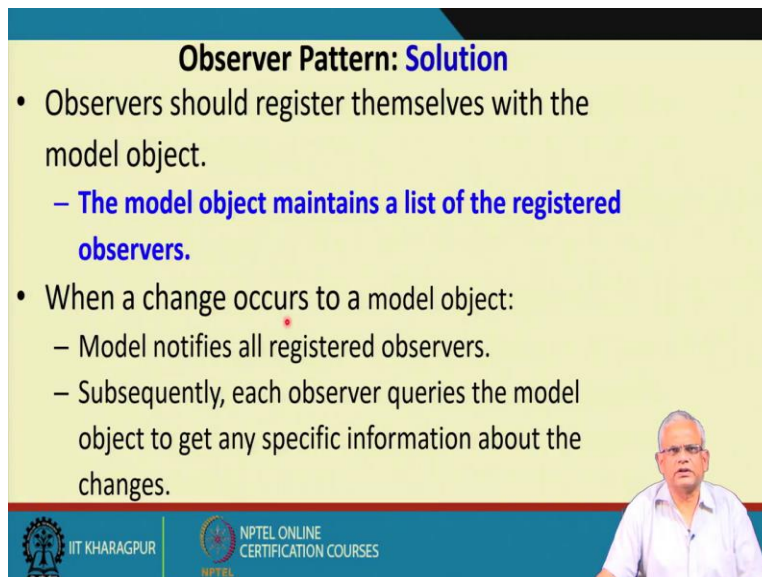
Observer Pattern: Main Idea

- When observable (model) changes state:
 - All dependent objects are notified --- these objects update themselves.
 - **Allows for consistency between related objects without tightly coupling the classes.**

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The main idea of the solution here is that the model keeps track of any changes that occur the model is also called as the observable and somehow it keeps track of the references of all Observers it does not hard-code but then the Observers register with the model by supplying the references and the observable at any time keeps track of all the Observers were interested in data and then whenever there is a change, it notifies them the objects then, the Observers then update themselves. We will see more details of the solution.

(Refer Slide Time: 17:12)



Observer Pattern: Solution

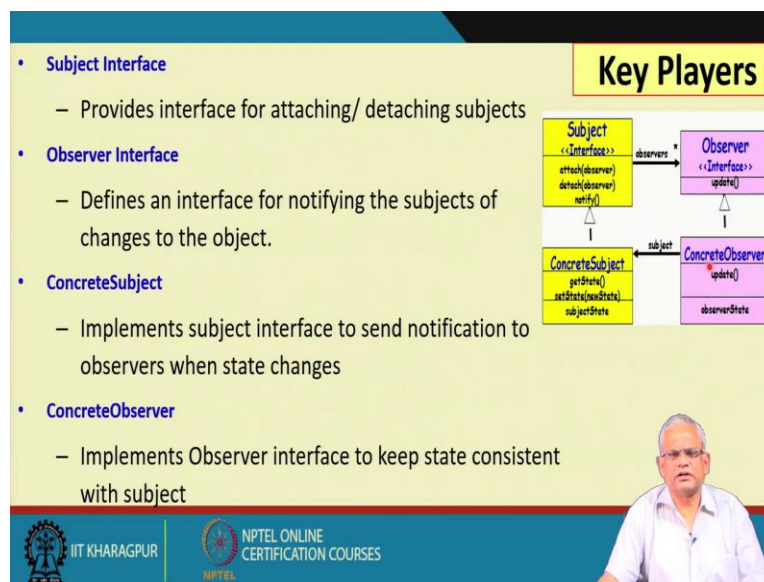
- Observers should register themselves with the model object.
 - **The model object maintains a list of the registered observers.**
- When a change occurs to a model object:
 - Model notifies all registered observers.
 - Subsequently, each observer queries the model object to get any specific information about the changes.

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The Observers registered themselves with the model by supplying their reference, the model intern maintains the Observer references. In the form of an array list whenever there is a model change to the model, the model notifies all the registered objects whose IDs appear in the array list and then the Observers once they know that there is a change, they can query specific type of change.

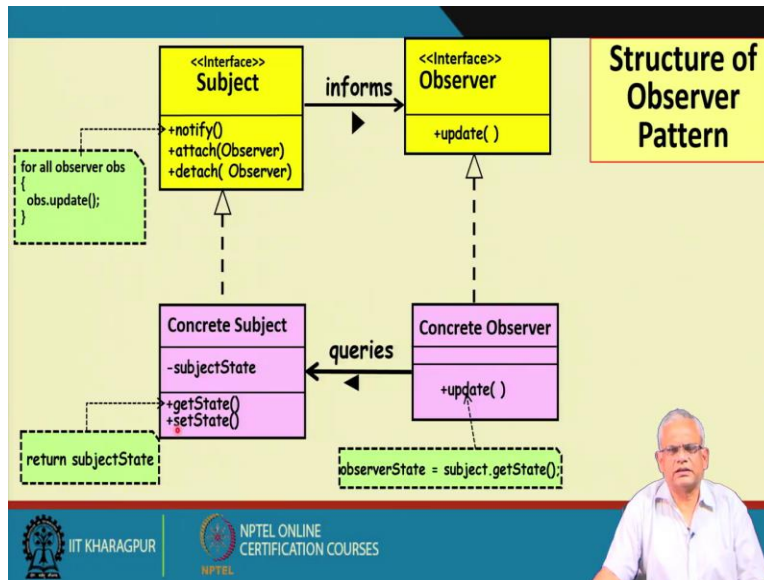
For example, in the stock market, the Observers may be notified that there is a change in the stock price but the specific customers might query about whether a stock price of A has changed another Observer may query whether the stock price of B has changed and so on that is the solution here.

(Refer Slide Time: 18:23)



In the form of a class diagram, we have a subject which is an interface. The subject defines the methods attach that is for the Observers to register detach that is when an Observer leaves calls the detach method and then notify, this is when a change occurs in the model it notifies all the registered Observers. The concrete subject implements the subject interface and on the Observer side we have the Observer interface and here there is an update method which is called by the notify method on the subject side to notify about the changes and the concrete Observer implements the Observer interface.

(Refer Slide Time: 19:26)



If we try to look at the code, in a notify the code there will be for all observers who have registered call `observer.update` and once the update method is called, the Observer might try to get specific information about the change that has occurred and for that it calls `getState`, the `getState` method body is only return the subject state and then the subject may change asynchronously by other model objects and so on and they just set state.

(Refer Slide Time: 20:27)

The slide is titled "Observer Pattern" and lists the following characteristics:

- Defines a one-to-many dependency between objects:
 - When one object changes state, all its dependents are notified and updated automatically.
- Decouples the subject from the observer:
 - Since the subject has little information about needs of the observer.
 - Can result in excessive notifications.

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

As you can see that by the Observers attaching themselves to the model a one-to-many dependency setup dynamically and when the model changes it notifies all the dependent objects.

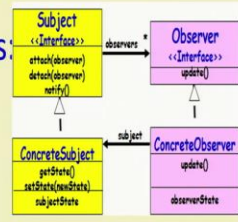
In this way, the Observer and the model are decoupled, but then the model just notifies all Observers that a change has occurred, does not keep track of what information is needed by which Observer, just calls the update method and the Observer and the Observers then get state of the subject or the model.

But a problem here is that, since there may be many Observers and each of them trying to find out what change has occurred. Just giving an example of the stock market and there may be large number of Observers who were interested in different stock price data, just saying that the market price has changed then every Observer tries to check whether his stock price has changed, this may result in excessive notifications. This we must be aware about the limitation of this Observer pattern, there are further adaptations and modifications to this pattern to take care of this situation.

(Refer Slide Time: 22:18)

Working of Observer Pattern

- ConcreteSubject notifies its observers:
 - Whenever a change makes its state inconsistent with the observers.
- After being informed of change:
 - A ConcreteObserver queries the subject to reconcile its state with subjects.*

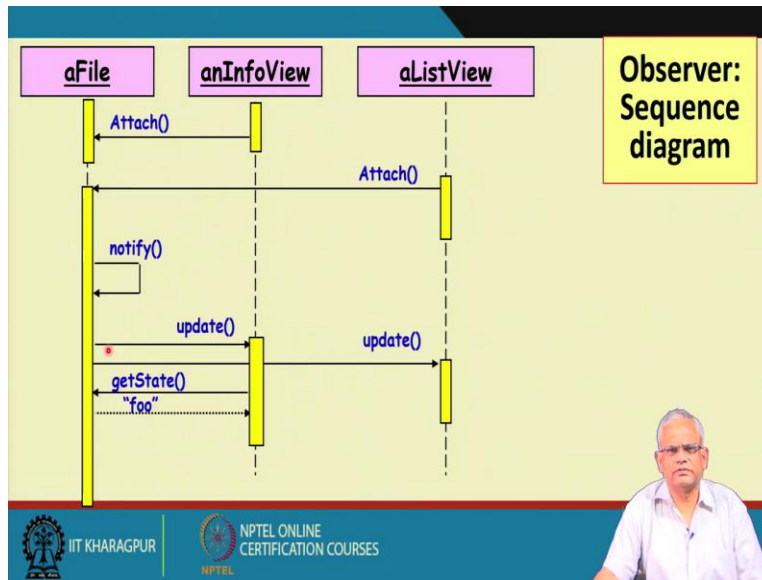


The diagram illustrates the Observer Pattern structure. It features four classes: Subject (an interface), ConcreteSubject (implements Subject), Observer (an interface), and ConcreteObserver (implements Observer). Subject defines methods attach(observer), detach(observer), and notify(). ConcreteSubject implements these methods and includes attributes getState(), setState(newState), and subjectState. Observer defines the update() method. ConcreteObserver implements update() and includes the observerState attribute. Relationships are shown as follows: ConcreteSubject inherits from Subject; ConcreteObserver inherits from Observer; ConcreteSubject has a collection of Observer objects (observers) and a reference to a ConcreteObserver object (subject).

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let us look at the working of the Observer pattern. The concrete subject notifies all the Observers who have registered and then once it notifies by calling the update method on the Observer, the Observer internally calls the getState.

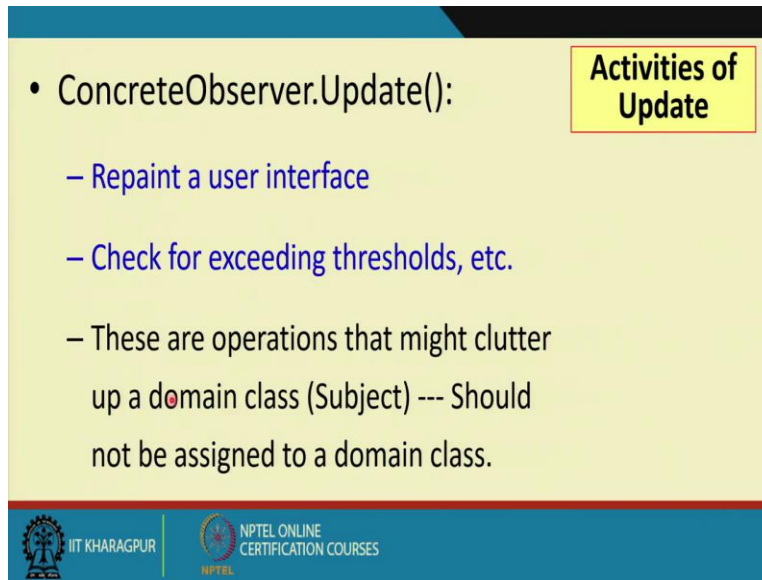
(Refer Slide Time: 22:45)



In the form of a sequence diagram, we have a file here which is the model and then there are several view object, one is an information view, this is all the details of the file size, date created that modified specific protections on the file and so on and there is a list view which just lists the file, there may be an icon view which just shows the different file icons and so on. We might have several views of the same file objects. Now, let us say the file size has changed.

Now all the views have to be updated for the solution we have different view objects attached to the file object and then as changes to the file locker like there is a save on the file, the user updates and shapes then the file notifies all the registered viewers during attach they would have provided their IDs and the file spot of the notify process calls the update and the different views and then the different views may `getState` and the state is returned to the Observers and that is the working of this Observer pattern.

(Refer Slide Time: 24:33)



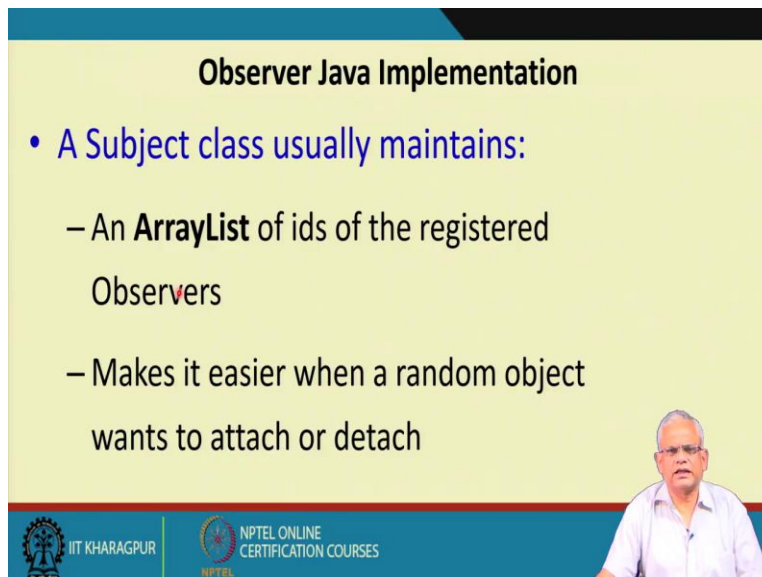
Activities of Update

- ConcreteObserver.Update():
 - Repaint a user interface
 - Check for exceeding thresholds, etc.
 - These are operations that might clutter up a domain class (Subject) --- Should not be assigned to a domain class.

Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

In the update method on the Observer, it might involve different types of activity. For example, it may just display the text or if it may show a graphics repaint a user interface or it may do some checking like threshold check and so on and these activities specific to the Observer need to be taken care by the Observer itself. It will be a bad solution, if the model takes care of this activity, a good solution should not do this, it just notifies the Observer and the Observer then takes care of the changes that are necessary.


(Refer Slide Time: 25:22)



Observer Java Implementation

- A Subject class usually maintains:
 - An **ArrayList** of ids of the registered Observers
 - Makes it easier when a random object wants to attach or detach

Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.



Now, let us look at the Java implementation of the Observer pattern. The subject maintains an array list of IDs. As the Observers register, the subject updates the ArrayList adds the reference of the Observer by keeping track them in the ArrayList it becomes easy to insert or attach or also delete the Observers.

(Refer Slide Time: 26:02)

```

public interface Observer {
    public void update(Subject o );
}

Public interface Subject {
    public void addObserver(Observer o);
    public void removeObserver(Observer o);
    public String getState();
    public void setState(String state);
}
                
```

Observer Pattern Java Implementation

Now the subject, the Observer is an interface and we have update method defined there. The subject is interface and we have addObserver, removeObserver, getState and setState. These are the method prototypes defined in the interface.

(Refer Slide Time: 26:31)

And then we have concrete subject that implements the subject interface or if it is an abstract class, the concrete subject is derived from the subject, it provides specific methods to different subject, for example, getState may return subjectState and if it is an abstract class, it may have method definitions. For example, for notify, it might have for all o in Observers o data update and in the update we have the Observerstate is equal to subject.getState.

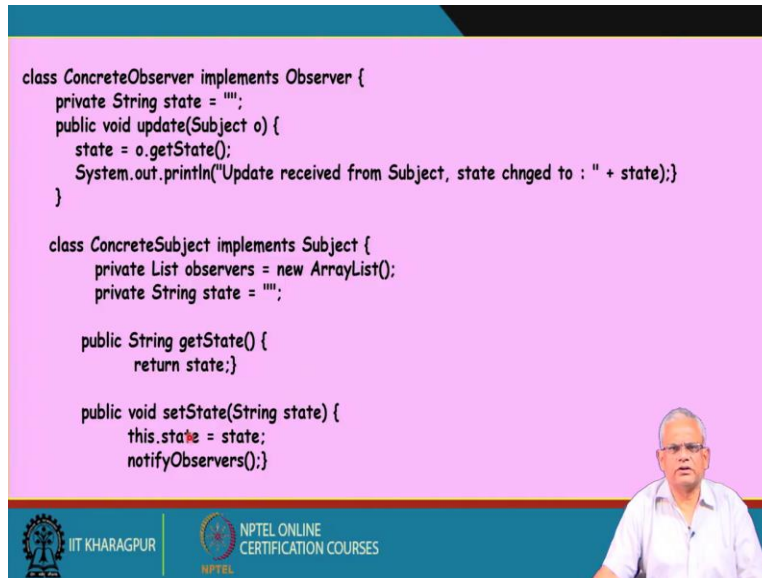
(Refer Slide Time: 27:13)

```
class ConcreteObserver implements Observer {
    private String state = "";
    public void update(Subject o) {
        state = o.getState();
        System.out.println("Update received from Subject, state chnged to : " + state);
    }
}

class ConcreteSubject implements Subject {
    private List observers = new ArrayList();
    private String state = "";

    public String getState() {
        return state;}

    public void setState(String state) {
        this.state = state;
        notifyObservers();}
}
```



This is the concrete code. The Concrete Observer implements Observer and it defines the update method. It keeps track of the state and this state should be synchronized with the state of the model or the subject whenever the model changes it calls the update method on the Observer, it calls the great state and it updates the state that it maintains.

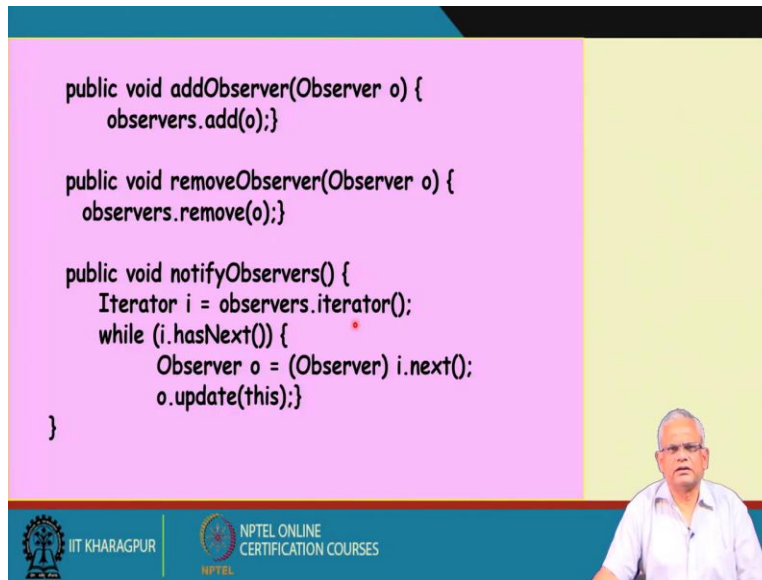
The concrete subject it implements the subject. It keeps track of the Observers in an ArrayList and it has its state, whenever there is a getState, it returns the state and the setsSate is called a synchronously by other objects, other model objects or other interface objects who just sit the state, unless the state changes it calls the notify Observers.

(Refer Slide Time: 28:28)

```
public void addObserver(Observer o) {
    observers.add(o);}

public void removeObserver(Observer o) {
    observers.remove(o);}

public void notifyObservers() {
    Iterator i = observers.iterator();
    while (i.hasNext()) {
        Observer o = (Observer) i.next();
        o.update(this);}
}
```



We have the oddObservers where it is added to the ArrayList, removeObserver removed from the array list and then notify Observers, for all Observers in the ArrayList, it just calls ordered update using iterator it traverses the ArrayList and calls the update method and all the elements of the ArrayList. We are almost at the end of this session. We will stop here. We have a small discussion left out on the Observer pattern. The Observer pattern is so popular that it is even implemented in the Java language itself, we will just look at that and then we will conclude the Observer pattern that is in the next session. Thank you.