

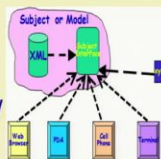
Object Oriented System Development Using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 43
Observer Pattern I


Welcome to this session. Over the last couple of sessions, we have been discussing about the Gang of four patterns of the GoF patterns. And we had said that we will initially discuss very simple, intuitive patterns which are applicable in a wide ranging problems, very popular patterns. We started looking at the facade pattern, very simple pattern. And then we had started discussing about the observer pattern. Now, let us continue our discussion about the observer pattern.



(Refer Slide Time: 01:05)

- **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?

Observer Pattern







The observer pattern addresses the problem when we have a model or a subject. The observers they want to get updated. Or they want to observe when changes can happen. On the model, the change to the model happens asynchronously. They may occur any time and therefore continuous pulling from the observers about if there is a change is highly inefficient. It is not a proper solution that the clients keep on pulling the keep on invoking the method on the subject class is not appropriate if the number of observers is fixed.

Like, let us say there are only two terminals on a computer and the changes need to be shown on the computer, then possibly the model object can show the changes directly because these are

directly coupled to this and they just need to invoke the method on those. But we are looking at a situation where the number of observers is not fixed.

New observers may arise. Some observers may drop out and the changes in the model cannot be predicted. We are just taking the example of a cricket game and there are many observers of the cricket game. Some are sitting on their desktop. Some are using laptop with various operating systems and so on. And some are using mobile phones etc. And these clients or the observers can join any time and drop out anytime. And we are looking at a good solution about how this can be solved. How the interaction between the model element and these observers are the view elements.

How the interactions can be structured? We have so far in all the discussions that we had, we said that in any application we will find the model objects or the entity objects. These store information and there may be some controller objects. And then there are the interface through which service requests may come. And we can have observers or the view objects. I think we will just write view, model view controller, so we will write view. The view objects and is it that the view objects directly invoke services on the model object? In that case, the view object will be coupled to the model object.

And also, the model object cannot really update the view because it does not know the view objects. They may come any time. And also, they may leave any time. And that is the problem we are addressing and the model objects, the change due to various reasons, maybe the cricket game, some event occurred, run scored and the model got updated or maybe in a networking, some message and the network model got changed. Or maybe in a stock market, the model is the prices of the various stocks in the market and some trading took place and the price of various stocks changed asynchronously.

We do not know when it will change, but then the value changes asynchronously. And then those who are watching the market behaviour, the market, they would like to see some specific stocks. What is a price? And whenever there is a change only that need to be notified. So, that is basically the problem that is addressed here.

(Refer Slide Time: 06:34)

Observer Pattern

- **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.

ARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now, let us look at the solution here, the solution offered by the observer pattern is that define a one too many dependency so that when the model changes state, all the dependents are notified and updated automatically. Let me elaborate the solution. There is one to many dependency. So, we have to set up this one to many dependency.

Somehow, we need to have this dependency one to many. These are the observers. And this is the model and there is a one to many dependency. So, that whenever there is a model changes, the observers will get notified, observer 1, observer 2, observer 3 and so on. But then the problem here is that the observers can come any time and go anytime. And for that, as we will see the details of the solution offered by the observer pattern.


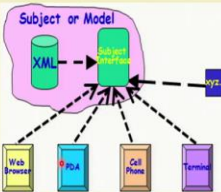
The observers need to first report to the model. That they are interested, and then the model stores, the ID of the observers, and when they leave, they just communicate to the model about living the observation. Many observers can attach themselves to the model, the model stores their ID and whenever there is a change, it communicates to the observers.

So, the solution is simple. There is a one to many dependency, not many to one, many to one would have been if the different observers here they would be keep on requesting the model at the subject.

(Refer Slide Time: 09:06)

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



IT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

There can be many observers and the observers vary dynamically, some may come and go, and the model changes state asynchronously. And also, the different observers may react differently to a change. For example, somebody maybe using a desktop. And there may be the actual match is being shown, the video is being shown and in some cases, maybe just the score is being shown and in some cases maybe only the message is being played that a run is scored. So, different observers, depending on the notification, may act differently. Some are just saying that matches scored, some is just updating a table and some playing a small video about how the run was caught.

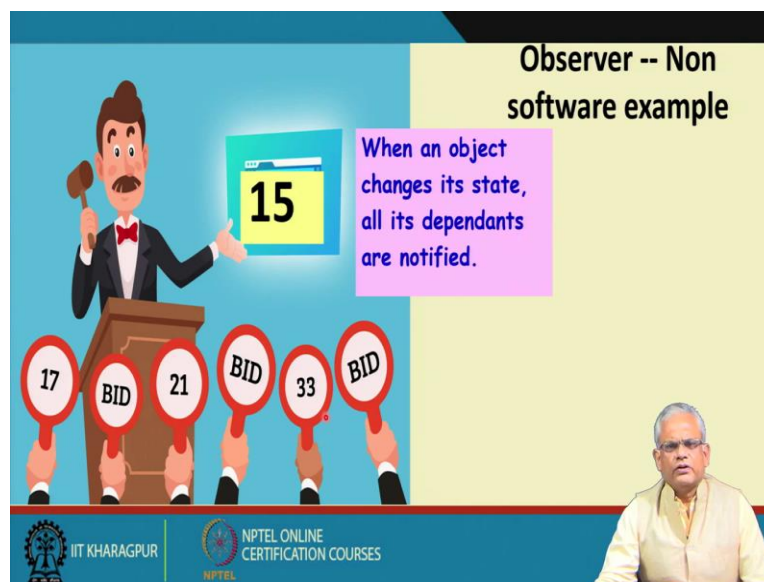
And the solution here offered is that the subject and the model should be decoupled from each other. That gives us flexibility to add more observers and also remove the observers. And we can also have additional observer types who can view the model changes. So, there are in summary, there are two or three things that we must find for the observer pattern to be applicable. The first thing is that the model must change asynchronously. If the change is Synchronous. That means predictable change, then the clients can directly call on the model element.

For example, if it is a daily update on the price next day only the price will get changed. Then the clients can invoke at every day beginning. In that case, the observer pattern is not applicable, the

observer pattern is applicable when the model changes asynchronously unpredictably for some time, there will be no change and sometimes some aspect may change and so on.

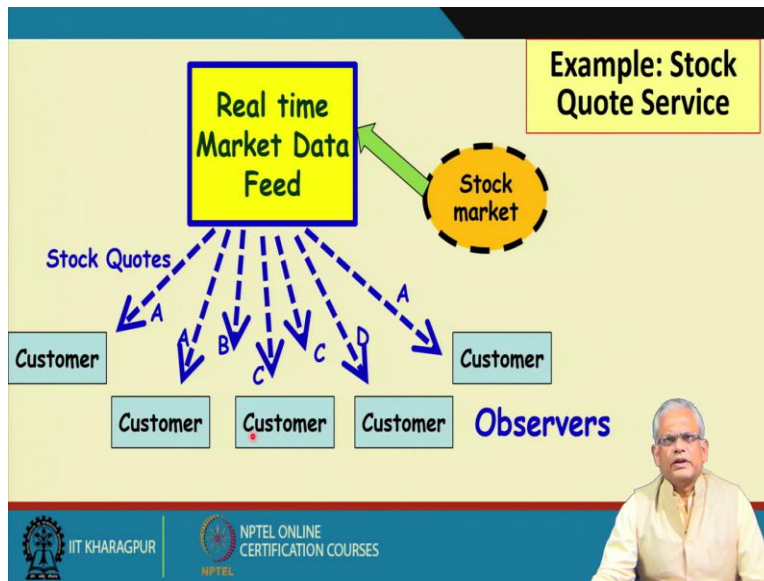
And the second thing for the observer pattern to be applicable is that the number of clients should vary dynamically. The type and number of clients vary dynamically i.e., with time, more observers may be interested. Some may drop out. That flexibility should be provided. And under these situations we use observer pattern. If the number of observers is fixed all the time, only those few observers or many observers there fixed then also this pattern is not applicable.

(Refer Slide Time: 12:34)



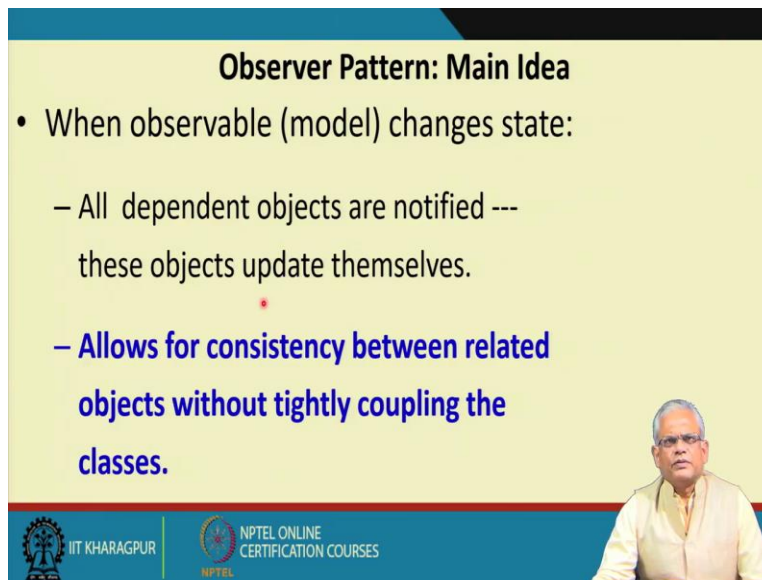
Here is a non-software example of the observer pattern. A bidding is going on and different clients are bidding different amount based on what they are observing. And they cannot see each other. And the model here is the state of the bid. The person bidding observes the maximum price is 33 and then updates it to 33. And as soon as the 33 is updated here, then everybody is able to see 33. This is an example of an observer pattern, non-software example, because here the model has a state which is the latest price that has been accepted. And the price can change any time to some amount, it can be 17 or 21 or 33 or something. But then as soon as the price is updated, then everybody can see the price. So, this is somewhat like an observer pattern that we are going to discuss. The accepted price can change any time, and whenever it changes, then the. Persons who are bidding are notified with the change of the price.

(Refer Slide Time: 14:26)



This is another example of the stock quote service. The market data is obtained, real time market data feed comes here to this class. So, this is the one stock market data is coming here, the market data, the state of the market is changing, and then there many observers. And more observers may come in and some of the existing observers may drop out and they may be interested in specific stocks. For example, customer A may be interested about A. Customer, this customer is requested about D customer. This is about C, this is also C and so on. Now, the problem here is that we not only need to have the customer notified of the change, but also the specific stock that changes. Now, let us see what is the solution offered by the observer pattern.

(Refer Slide Time: 15:44)




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

NPTEL




Here the main idea is that the different observers. We will first request the model that they are interested in some behaviour. The models are the observable, will register them, and as soon as there is a change, then all the dependent objects who have registered are notified. And since here the dependency set up and broken as the request from the observer comes to get attached to the model. And the model registers them and whenever they want to go away, they just send a request and the observer decouples them. This prevents tight coupling between the classes.

(Refer Slide Time: 16:50)

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.



Logo of IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

The solution offered here is that the observers request the observable to register them. The observable stores the reference to the Observer during the registration. And when there is a change to the model, the model notifies all the registered observers. But then it just notifies that there is a change. And different observers might be interested in different aspects of the change.

For example, some observers may be interested in the stock price of an other observers, maybe requested in stock price B, C, etc. Now, once the observer is notified of a change by the model, then they can query from the observer the specific price of the stock they are interested.

(Refer Slide Time: 17:55)

Key Players

- **Subject Interface**
 - Knows its observers – provides interface for attaching/detaching subjects
- **Observer Interface**
 - Defines an interface for notifying the subjects of changes to the object.
- **ConcreteSubject**
 - Implements subject interface to send notification to observers when state changes
- **ConcreteObserver**
 - Implements Observer interface to keep state consistent with subject

```
classDiagram
    class Subject {
        <<interface>>
        +attach(observer)
        +detach(observer)
        +notify()
    }
    class Observer {
        <<interface>>
        +update()
    }
    class ConcreteSubject {
        +getState()
        +setState(newState)
        +subjectState
    }
    class ConcreteObserver {
        +observerState
    }
    Subject <|-- ConcreteSubject
    Observer <|-- ConcreteObserver
    Subject --> Observer : observers
    ConcreteSubject --> Subject : subject
    ConcreteObserver --> Observer : update
```

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

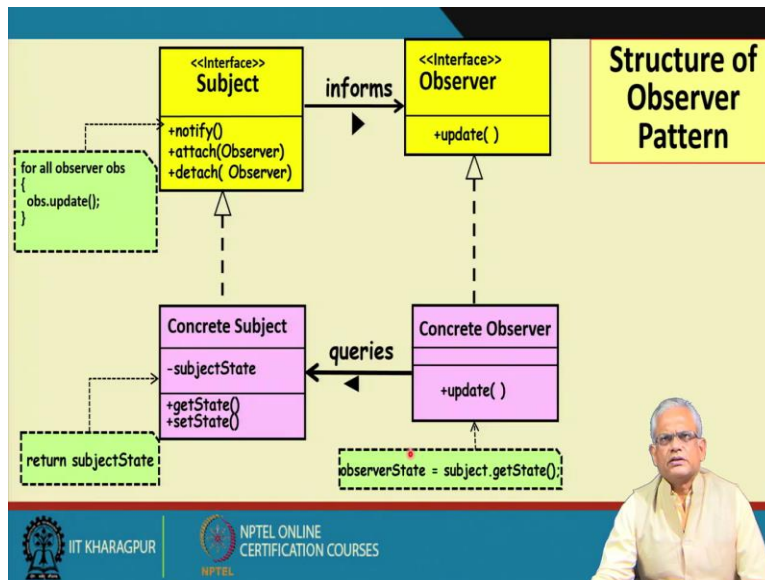
Now, this is the class structure recommended by the observer pattern, as you can see that there are four classes here. One is the subject interface. This is the interface class. And we know that interface is a good mechanism to decouple and reduce coupling, to reduce dependency. The subject interface provides prototype methods for the observers to attach and detach. And also, it has method to notify and the concrete subject implements the subject. And it can get state, that is, it can notify the clients when a state change occurs. And there may be other classes which are trying to set state of the concrete subject that are not seen, just like the stock market.

The market may call the concrete object set state for when a specific change occurs. And now there are observers which have attached to the subject, and then as soon as change occurs and the concrete subject notifies the corresponding observers. And then the observers can update themselves based on the notification, so that is what there are four classes, the subject interface, the observer interface, these help to decouple the concrete subject. And the concrete observer from each other. We had seen so far how this principle works program to an interface.

The concrete subject implement the subject interface and sends notification to the observers, the concrete observer implements the observer interface and provides a definition to the update method and how it can be updated. For example, whether a table is shown about the runs,

whether a video clip is played, whether an audio clip is played and so on. These are taken care by the specific implementation of the update method of the concrete observer.

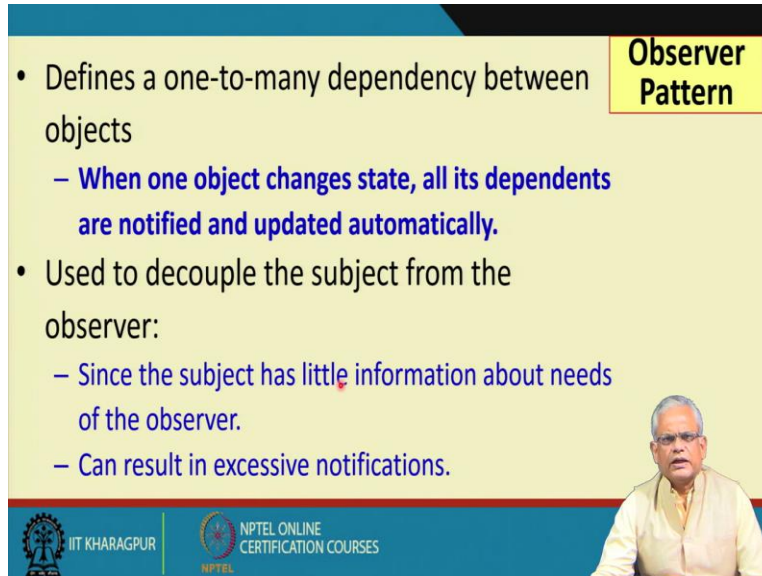
(Refer Slide Time: 20:50)



Now, to understand the working little better. The subject, the concrete subject stores the references of the various concrete observers after the Observer attaches their store and as a change occurs, it calls the update method on the concrete observer whenever there is a change, the concrete subject calls the update method of all the registered observers. That is what shown here. The notify method of the observer contents, for all observers that have registered invoke the `observer.object`. And based on this, the concrete observers are notified.

But then they may be interested in the update, they may be interested on specific aspects of the change and they may query the concrete subject, `subject.getState`. And then it returns the state here and the concrete observer state refreshes itself. For example, the concrete observer may be interested in some specific stock price. And once it is notified that there is a change, it just gets the corresponding stock prices.


(Refer Slide Time: 22:38)



Observer Pattern

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

IT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES



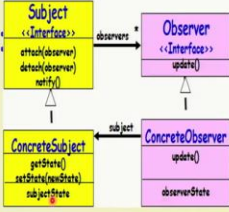
As you could see that there is a one to many dependency set up through the registration process attach. And then when the subject changes, it notifies all the dependent or attached observers. But a problem with observer pattern, if we think of it, that even though the observer is not interested in the stock price of company A, but then he has got a notification that the price has changed and when invokes the getState, finds that A has changed.

Whereas the observer is interested in C. So, there is an unnecessary notification, excessive notification. This may be a shortcoming, but then we can overcome it in various ways. For example, we can just modify the observer pattern little bit and have the observer. Also determine for which in event. A specific client or observer is interested and then only for that event notify that client, but that makes the model more complicated.

(Refer Slide Time: 24:21)

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.



```
classDiagram
    class Subject {
        <<Interface>>
        attach(observer)
        detach(observer)
        notify()
    }
    class ConcreteSubject {
        getState()
        setState(newState)
        subjectState
    }
    class Observer {
        <<Interface>>
        update()
    }
    class ConcreteObserver {
        update()
        observerState
    }
    Subject <|-- ConcreteSubject
    Observer <|-- ConcreteObserver
    Subject "1" -- "*" Observer : observers
    ConcreteSubject --> ConcreteObserver : subject
```

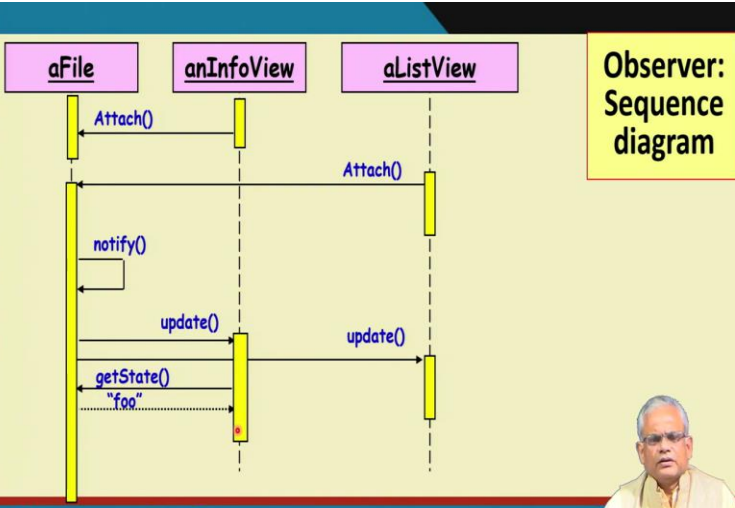
The diagram shows the Observer pattern structure. It includes a Subject interface with methods attach(observer), detach(observer), and notify(). ConcreteSubject implements these methods and has attributes getState(), setState(newState), and subjectState. The Observer interface has an update() method. ConcreteObserver implements update() and has an observerState attribute. Arrows indicate that ConcreteSubject inherits from Subject and ConcreteObserver inherits from Observer. There is an association between Subject and Observer labeled 'observers', and an association between ConcreteSubject and ConcreteObserver labeled 'subject'.

Logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES are visible at the bottom.

Now, the concrete subject as the changes occur notifies the concrete observer after the changes in informed. The concrete observer queries the subject to find what is the state that has changed and it changes the state. Makes display the correct state and they are synchronized.

(Refer Slide Time: 24:52)

Observer: Sequence diagram



```
sequenceDiagram
    participant aFile
    participant anInfoView
    participant aListView
    aFile->>anInfoView: Attach()
    aFile->>aListView: Attach()
    aFile->>aFile: notify()
    aFile->>anInfoView: update()
    aFile->>aListView: update()
    anInfoView->>aFile: getState()
    aFile-->>anInfoView: "foo"
    style aFile fill:#f9f,stroke:#333,stroke-width:1px
    style anInfoView fill:#f9f,stroke:#333,stroke-width:1px
    style aListView fill:#f9f,stroke:#333,stroke-width:1px
```

The sequence diagram illustrates the interaction between aFile, anInfoView, and aListView. aFile calls Attach() on both anInfoView and aListView. When aFile calls notify() on itself, it then calls update() on both anInfoView and aListView. anInfoView then calls getState() on aFile, which returns the value "foo".

Logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES are visible at the bottom.

If we represent the same thing in the form of a sequence diagram, initially, the observers of some file here, the file has many observers. One is let say information view that it provides icons and

another is list view. The list view only lists the file and the information view, it provides detail of the files, size and so on. Here it just lists.

Now, let us say the different observers, the first attach. The file keeps track of these different views of the observer. And then as a change occurs to the file, it calls the notify method on itself. And then the different observers are the update method is called on the different observers and then the observers may request the getState. It returns the state and then they change their own state. And the observer and the model gets synchronized.

(Refer Slide Time: 26:30)

• ConcreteObserver.Update():

- Repaint a user interface
- Check for exceeding thresholds, etc.
- These are operations that might clutter up a domain class (Subject).

Activities of Update

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Once the update of the observers is called the observer may carry out different activities, for example, it may repaint its interface or it may, check whether some threshold exceeded. And the subject, if we had to do that, if the model had to do this, that on an event what things should take place on the observer. That will make a model extremely complicated.

So, it just informs that something has changed by calling the update method of the corresponding observers and the observers take care of what things to be done specific to them. This is the main idea of the observer pattern. And we are almost at the end of this session. We will stop here and we will have a couple of more examples. And also, the Java code to illustrate the observer pattern, it is important pattern.

So, important that in the Java language itself, the observer and observable interfaces are inbuilt. And we will see the code specific code how to set up an observer pattern for specific examples so that given another problem, will be able to apply the observer pattern. We will stop here and continue in the next session. Thank you.