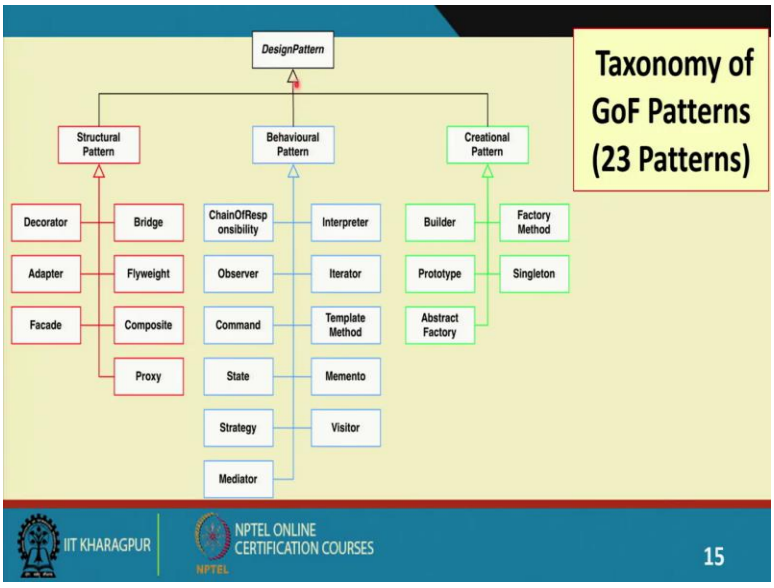
Object - Oriented System Development Using UML, JAVA And Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 42
Façade Pattern

Welcome to this session! In the last session, we had a brief introduction to the Gang of Four patterns popularly known as the GoF patterns. And we said that these are good design solution to some specific problems. And unlike the grasp patterns, here the problems are very specific and also the solutions are very specific in terms of class diagrams, Java code and so on. We mentioned that there are 23 design patterns and these are classified into three main types, but in all these 23 patterns. The main aim is to improve the design by having better cohesion, reduced coupling, encapsulation, reduction in dependency and so on. Now, let us look at the first pattern today.

(Refer Slide Time: 1:38)



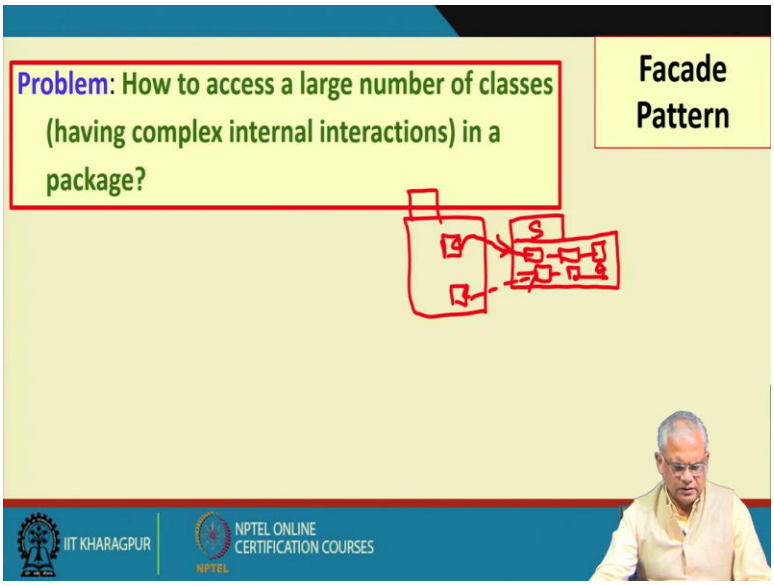
The 23 GoF design patterns are classified into the structural patterns, here how the interactions among the classes need to be structured. What are the specific classes to solve a well-defined problem and how the communication between these classes to be structured? That is the main focus of these patterns, and we will look at the decorator, adaptor, facade, bridge, flyweight, composite and proxy.

The behavioural patterns they are basically focused in realizing some behaviour and here we will discuss the observer, command, state, strategy, mediator, visitor, memento, template method, iterator and interpreter. And on the creational patterns, this concern with creation of objects. Usually complex objects so that it has reduced dependency and better creational operation.

And we will discuss here the factory method, a singleton, builder, prototype and the abstract factory patterns. The way we have structured in this discussion is that we will start with very simple patterns which are popular in the sense that useful in wide variety of applications and many applications.

And also very simple to spot and apply. So, in a sense, these are very useful patterns and the simplest we can straightaway start using them for solving specific problems. In this session we will start with the facade pattern, which is a structural pattern, very simple pattern and widely applicable. You might come across a code and you think that why the author did this in some way. But then after knowing the facade pattern, you can see why the code was written that way. And also for solving some specific problem, you will yourself use the facade pattern. And after discussing the facade pattern, we will discuss a behavioural pattern, which is the observer pattern. Now, let us get started with the facade pattern, very simple and useful pattern.

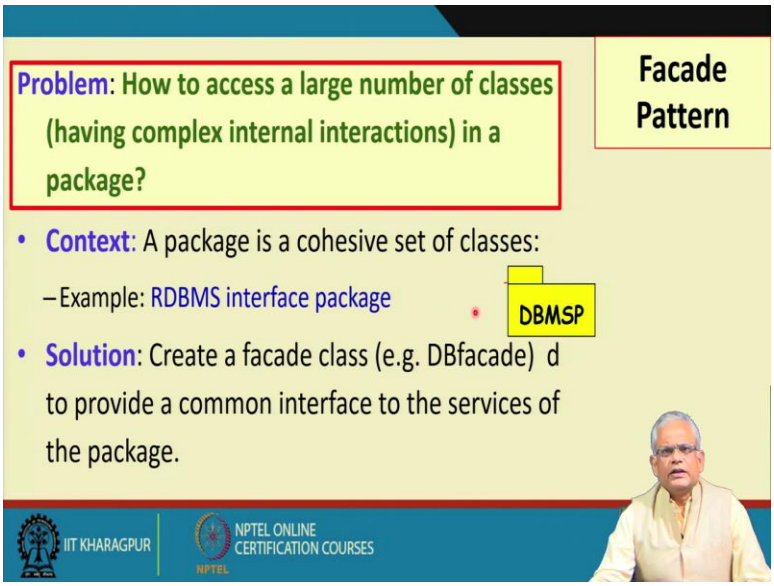
(Refer Slide Time: 4:37)



The facade pattern, it tries to solve the problem when a package has a number of classes inside it and the classes that are external to this package are trying to make use of the service of the elements in the package. So, let me just be more specific. This pattern concerns a service package. This is a service package written S here in the figure drawn above, and this has many classes inside it. And possibly these classes are related through inheritance, association, aggregation and so on.

Now, the classes that are outside this package may be belonging to other packages and so on. C1, C2 and so on. Now, how do these classes invoke the services? Invoke the services offered by the classes inside the package, service package. So, that is the problem, how the classes external to a service package invoke the classes inside the service package. And of course, the classes inside the service package might have complex interactions among themselves. And it should be totally transparent to the classes which are invoking the service. That is the main idea of this pattern.

(Refer Slide Time: 6:53)



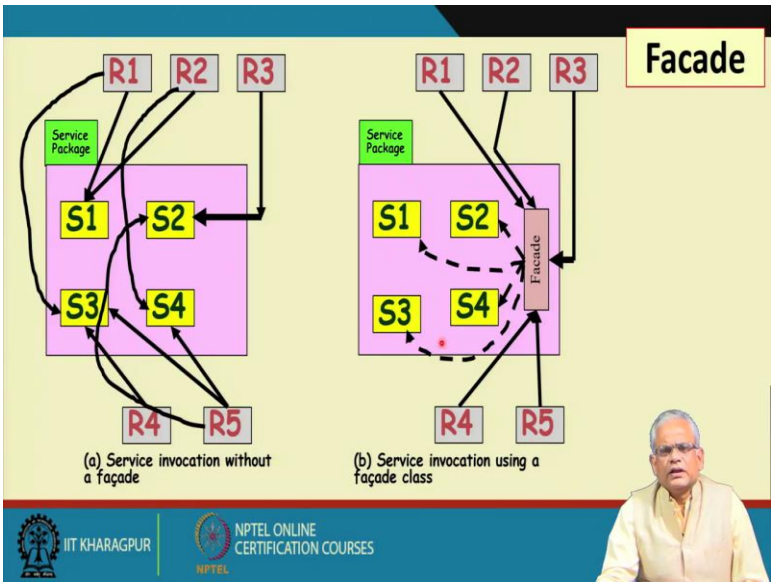
Now, here the service package is a cohesive set of classes, an example can be an RDBMS interface package where we might open connection to RDBMS, request various types of data and various types of operations on the data. And this is the package, this is a service package. The RDBMS service package and there are many classes inside it. Not shown them here, but there are many classes which collaborate among themselves to provide the necessary service. Now, the solution of the facade class.

The solution of this pattern is to create a facade class. We need to create a special class name of the class is facade. For example, for this package, we might create a DBfacade class to provide the interface for the service of the class. In other words, the external classes will not interact directly with the classes that are present in the service package. But they will interact only with a facade class. And the facade class, in turn, will understand the service requirement and will coordinate the activities or the services to be given and then provide the service.

But let me just ask this question that what is the English meaning of the word facade, because if we know the meaning, then the pattern purpose becomes more meaningful. If we just consider as a noun or something, it is not proper because it has a meaning. If you look at the dictionary facade stands for the frontal look of a building or something. The facade is not the actual thing, but the frontal appearance, the actual thing maybe something.

Maybe a person is wearing a mask of a ghost. The person is different, but he is wearing a mask and the mask, we can call as a facade, the facade gives a frontal appearance, which may be different from the actual thing and the same thing here. The facade is the appearance of the package to the external clients of this service package. Let me just repeat that thing that the facade class that we create inside a service class is the appearance of the package that the other client classes get the about this package. But then internally, something else may be happening.

(Refer Slide Time: 10:25)



Now, let us just illustrate it little bit. Let us say we have a service package. And the service package has various classes inside it, which might be requesting services from each other, cooperating and collaborating in various ways. Now, the external classes might just invoke the services. As they like, now which solutions are typically like this, that the external classes, they just request the service of whichever they feel like these are the client classes R1, R2, R3, R5.

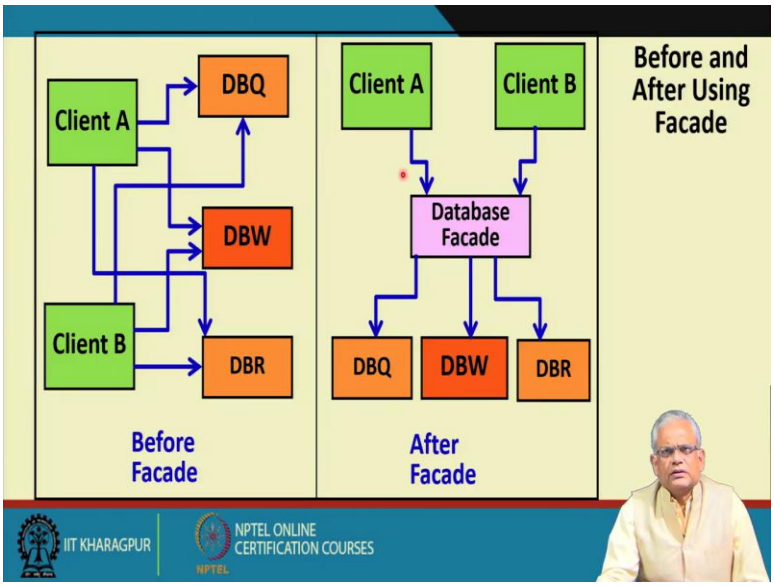
And the server classes are S1, S2, S3, S4 and they provide some specific service, maybe a database service, a network service or something. And then, R1 is invoking the service provided by S3. And also R1 needs the service provided by S1. But R3 needs the service provided by S2 and R5 needs the service provide S4 and that directly invoking the services.

But the solution provided by this pattern is that we should have a facade class. We should create a synthetic or artificial class, name of the class is facade. And then all the external client classes should invoke the services of the various classes inside the service package by requesting the facade class. When they make a request to the facade class, the facade class internally invokes the methods of the various server classes as required. Now, the solution is elegant, that can be immediately inferred from this diagram.

Look at how messy is this diagram. High coupling of the client with the server classes. If tomorrow the server changes then all these clients need to change, which are using it. But here they have been decoupled and the coupling of this client classes is only with a facade class. If the servers change, then most of the changes are observed by the facade and the clients are not even affected.

Now, let us look at some more aspects of this pattern, very simple pattern, simple idea, but very elegant idea that whenever the service of the classes inside the service package is required a client should invoke the requests on a facade class rather than directly invoking it on the server classes.

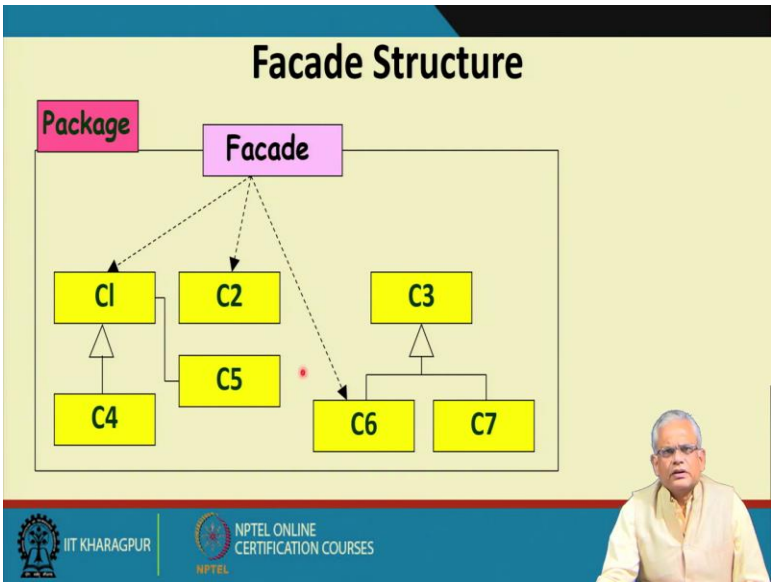
(Refer Slide Time: 13:49)



Now, let us look at a solution. Specific solution, these two client classes, invoking the database services database, read, write, query, etc. These are the classes which provide the service and this is a novice solution without using the facade. And then if we have a facade class, then this is the frontal of the package and this is only visible to the client classes. The other classes which actually provide the service is not visible to the client classes.

The client classes can only need to be associated with only the facade class. They need to have the reference or ID of the facade class. They do not have to have the coupling with all these classes. The dependencies are reduced significantly. Many of the changes in the server classes are not percolated to the client classes and the design becomes easy to understand.

(Refer Slide Time: 15:05)

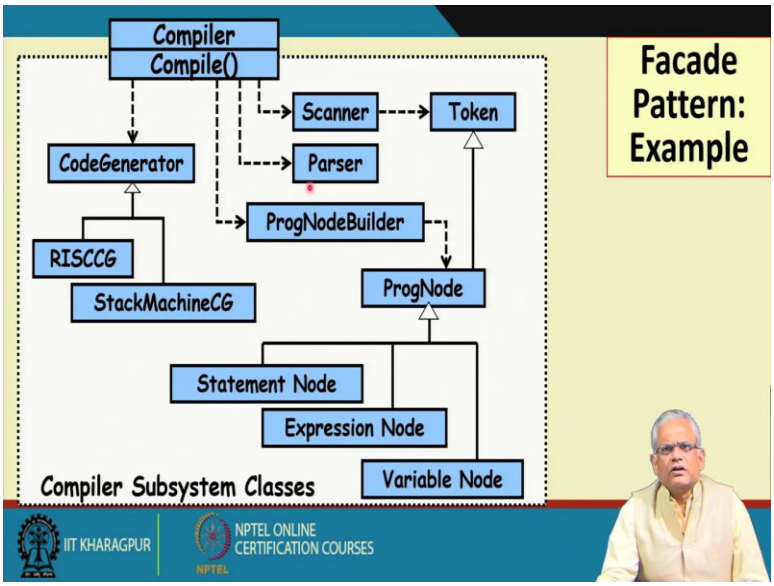


Typically, this class for every service package, we have to create a facade class. And the internally the different server classes might be related various ways among themselves. For example, inheritance here, there is association here, there is an inheritance here and so on as shown in the figure.

Now, depending on the request of the client, the facade class knows which classes to contact and get the result. The client classes would request some specific things, for example, they might need connection to a certain database. They might need to access or read some type of data, they might have to write some type of data, they might have to have a query of some types of data and so on.

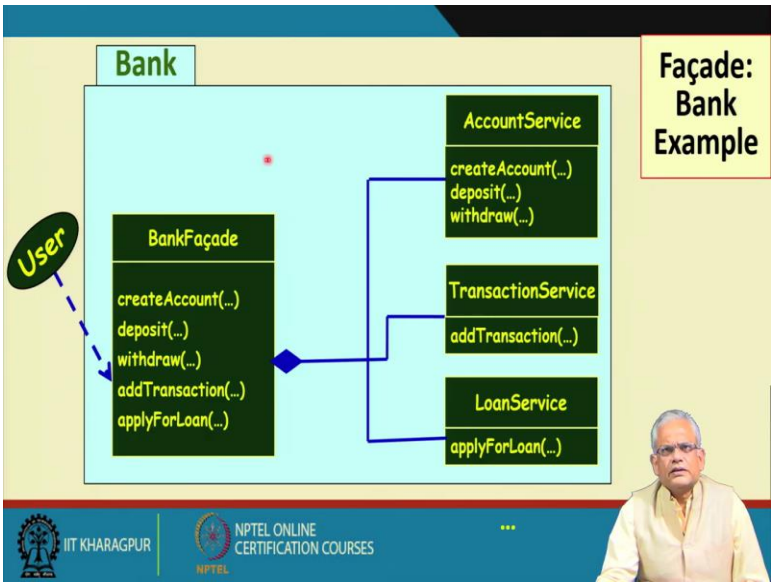
They just place the request with the facade and the facade does whatever is appropriate. The facade stores the IDs or is associated with all these service classes, various service classes it is associated. And it knows how to handle a request, maybe a request would require getting service from multiple classes and the facade will do that. In this situation, if we did not use the facade, the client class would request the specific services from these server classes.

(Refer Slide Time: 16:48)



This is another example; this is the compiler. Different classes are inside the compiler package. These packages, these hierarchy deals with code generation, there is a lexical analysis parsing etc. Now, the client will just say compile. Which is basically a method of the facade, the name of the facade here is compiler, so compiler.compile. That is enough. The client does not have to say that first do this lexical analysis, then do the parsing code generation, etc. It helps the trouble. The facade observes all that to just call the compiler.compile. And that takes care of contacting all these necessary classes in specific order and get the thing done.

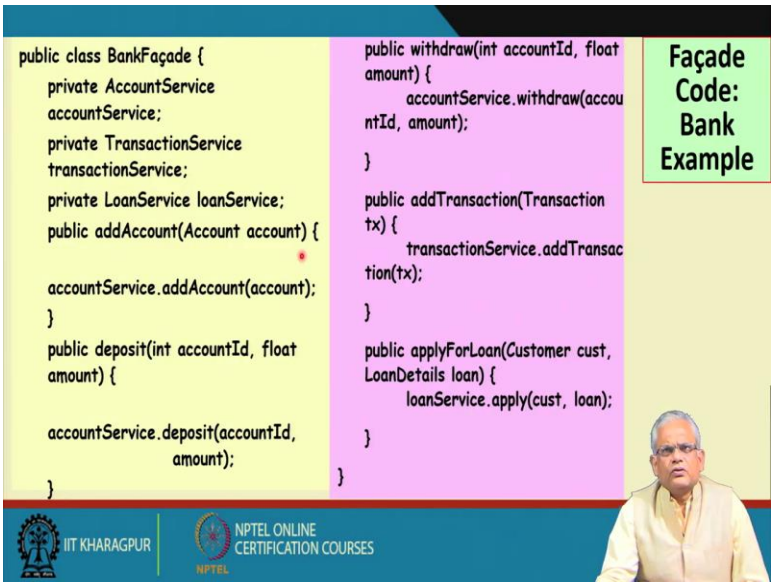
(Refer Slide Time: 18:02)



An another example here. This is a bank package. And then the bank provides various services to the clients, and for that the bank has created specific classes. The account service class. This may be package or class. The account service class, transaction service class and loan service class. Now, the bank package has as per the facade pattern has this bank facade class. And see here, it provides various services like creating an account, depositing, withdrawing, add transaction, apply for loan.

All services provided by the bank façade, user just knows the facade class and invokes the specific service apply for loan. And then the facade class has the ID for the loan service and then requests the corresponding method in the loan service class. Similarly, add transaction will invoke a service on the transaction service class and deposit, withdraw, create, etc. will be the facade will invoke the corresponding service on this class. Next time you read a code and find that there is a class named as BankFacade. You will understand that they have applied the facade pattern for a service package.

(Refer Slide Time: 19:52)



Now, let us look at the code for the bank example. This is the bank façade, this is the Java code for bank façade as shown in the figure above. And just see that internally, the facade class keeps the references of all classes or all the relevant classes inside the service package. The account service, the transaction service, loan service, and then as the request comes from the client to add account, then here it just invokes the corresponding method of the account service reference that it has stored internally. The account service reference that it has stored internally it just invokes that on the add service, accountService. addAccount.

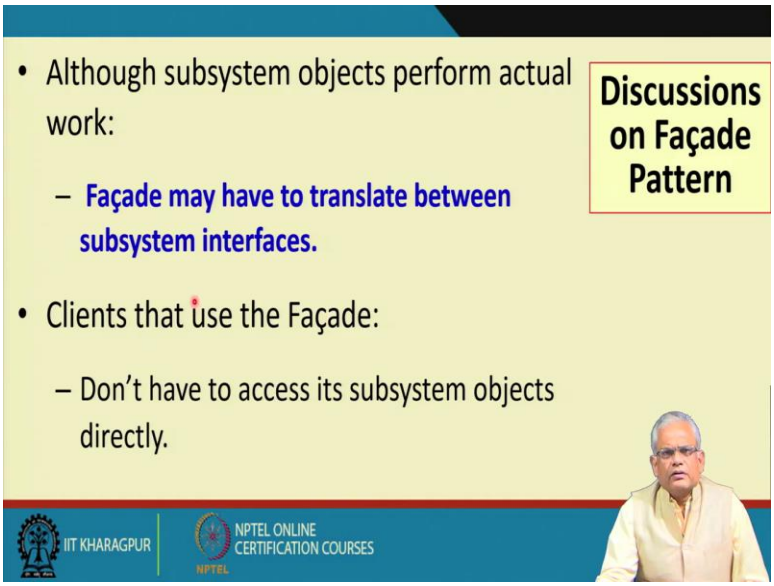
Similarly, when a deposit request come from the customer or the client, the client knows only the bank facade invokes the deposit method on the bank facade. And then internally it invokes the deposit method on the account service class. Withdraw request comes from the client and accountService.withdraw, withdraw is invoked on the account service class. Similarly, the transaction.addTransaction is invoked on the transaction service class. The apply for loan once the request comes from the user, the bank facade invokes that on the loan service class.

Basically, the clients keep track of only the facade and give it request and internally it translate that request into request to some classes at the simplest case. Now, we will see that there can be more to a facade class than just translating the request to specific service classes. For example, in the compiler, the facade not only requested the service, but it knew which services, sub services to request in what order and so on.

(Refer Slide Time: 22:20)

Now, if we think back. In the façade pattern, the clients communicate with a package. The classes in the package, by communicating only with the façade, the facade forwards this request to appropriate objects in the class, promotes low coupling. The clients need to be associated only with the facade. They do not have to have dependency on other classes. But then many times the facade does not do anything more than just delegating the request to other packages. But then the facade can also do things like it can have some business rules and for a high level request can break it down into smaller requests and get it serviced by appropriate objects in the package.

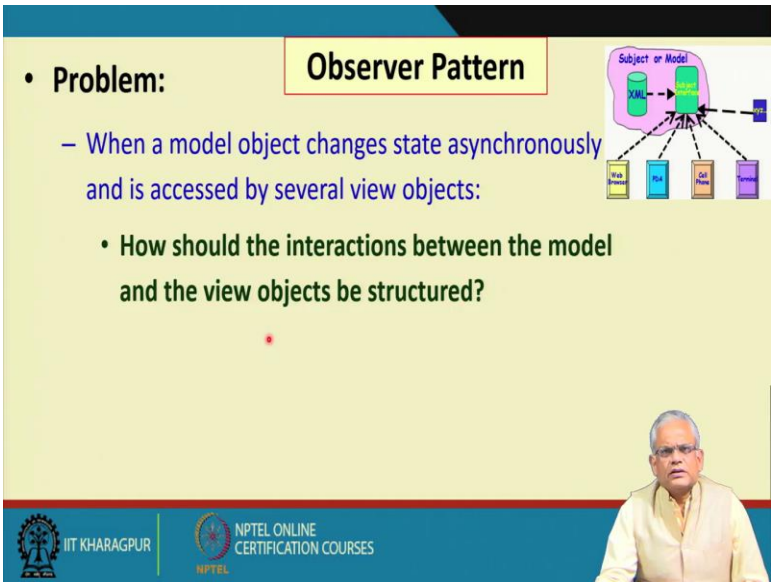
(Refer Slide Time: 23:24)



Another benefit of the facade that the users may invoke the request in various formats. And the facade may translate between the format with which the client requests and the request, the format in which the service class requires the request and this aspect provides for changing the service class without affecting the client.

The dependency between the service class and the client class is reduced because even if the server interface changes, the facade can observe the change here and only translate between the client request and the server request. And the clients of a service package do not access the classes in the package, they only interact with the facade class.

(Refer Slide Time: 24:32)



Now, let us discuss about the second pattern, second GoF pattern, which is the observer pattern. The facade is a very simple pattern and yet very useful. Whenever we are designing a service package, facade class must be present to reduce the coupling and improve the design. Now, let us look at the observer pattern, which is also a very simple pattern. It is a behavioural pattern. The observer is a behavioural pattern and here the main problem is that when a model object changes state asynchronously and is accessed by several view objects. How should the interactions between the model and view objects be structured?

So, here there is an observer and the model. There is a model here and there are observers. And the observers invoke the request to be refreshed with changes in the model. But the changes occur asynchronously. Now, how the interaction between the observers here and the subject needs to be structured? Just to give an example, let us say we have a bidding going on or let us say there are stock market quotations or most simple example, maybe a cricket match going on. And then people are watching the cricket match in various forms.

There are many observers of the cricket match. Watching remotely, some are sitting in front of their desktop using a Web browser. Some are using a mobile phone. Some are using an iPad and so on. And some are just listening to the commentary and so on. Now, they do not have to be updated continuously. Because that will make the system inefficient only when some changes occur like a run is scored or somebody is out and so on, a communication should occur.

And they should be updated about the status of the game because the status of the game has changed. This is an asynchronous change because in a Synchronous change that it occurs on a time step, it becomes easy for the client to query and get updated on the result. But here it can happen any time, maybe for quite some time there was no running. And suddenly a run was scored and only in that case they should be updated. And this pattern addresses a very elegant and efficient solution to this. And the other characteristic here is that the clients are variable.

They are not fixed clients. If it is a fixed client, then it can just invoke the corresponding objects. But then here the clients are, the client objects are not fixed. Somebody may just come in and start watching and somebody may just drop out and stop watching. We will look at the observer pattern solution in the next session. It is a very useful pattern and the solution is elegant. But we are at the end of this session. We will stop here and continue in the next session. Thank you.