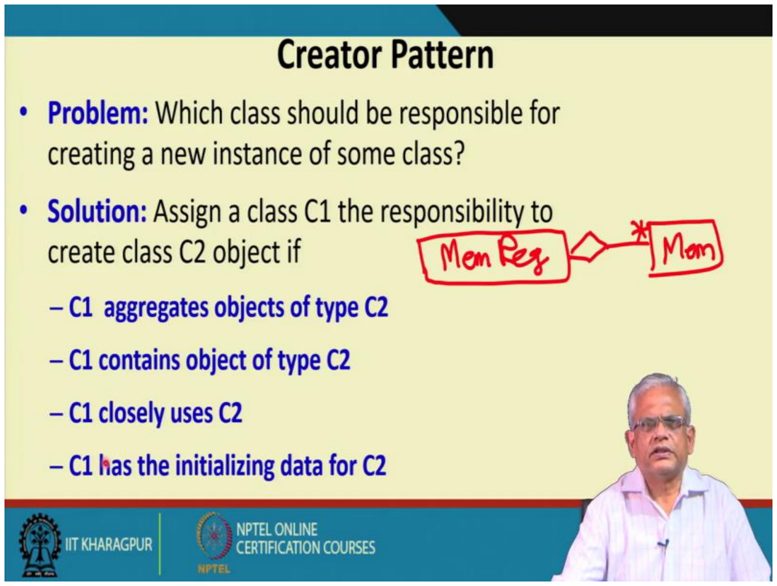
Object-Oriented System Development using UML, JAVA and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 40
Pure Fabrication, Law of Demeter

Welcome to this session! In the last session we were discussing about the GRASP patterns. Please remember that we had said that the GRASP patterns are extremely intuitive, very simple patterns, but then very highly usable. Based on simple design ideas, which are widely applicable in almost every design that we do. In contrast to the GoF patterns, which we will discuss shortly, in the GRASP patterns we just give the general idea here. That is how the pattern is described.

But in GoF patterns, we discuss this in terms of specific class diagrams, generic solutions described by specific class diagrams and code. But here, these are more general purpose and intuitive. And many of these, we do not even have a sample class diagram. We had looked at a couple of patterns, we had looked at the expert and creator pattern, and let us proceed from there.

(Refer Slide Time: 1:45)



Creator Pattern

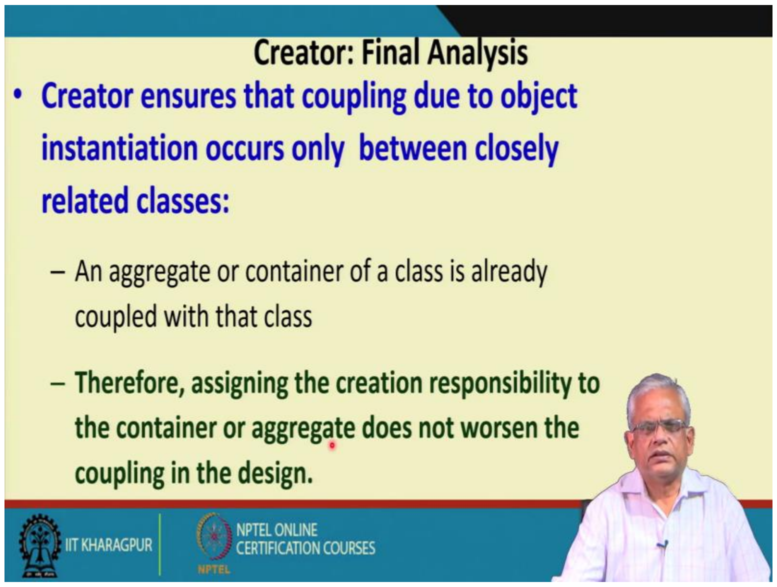
- **Problem:** Which class should be responsible for creating a new instance of some class?
- **Solution:** Assign a class C1 the responsibility to create class C2 object if
  - C1 aggregates objects of type C2
  - C1 contains object of type C2
  - C1 closely uses C2
  - C1 has the initializing data for C2

The creator pattern, which we were discussing last time, the problem occurs in almost every design that we attempt. The problem is when a new object is to be created, which class would be responsible for creating the object. And the solution here, we have specific solution. There are a few patterns in the GRASP pattern where we do not have very specific solution general ideas, but here we have specific solution that if we want to create an instance of class C2, then we need to give the responsibility to the class, which aggregates the objects of type C2.

So, if we have a member registered, which aggregates member records, then if we have to create a member, it should be done by the member register. In any case, the member registered will have to store in reference to the member object. And if it also creates the member object, that will be very simple to create it and also store the ID of the member register. And similar is the case when there is a composition, both aggregation and composition, we will have the aggregator, the composite creating an instance of the component, this is the second rule in the solution.

The third one is, if in case we have an object to be created, that is not aggregated into a larger aggregate object, then we can create such an object by a class which works closely with this subject that invokes methods or provides initialization data and so on. So, these two, if it is not the aggregate object, just an isolated singleton, then we should create or the creation should be the responsibility of a class, which either invokes a lot of methods in this class or it provides initialization data for this object.

(Refer Slide Time: 5:07)



Creator: Final Analysis

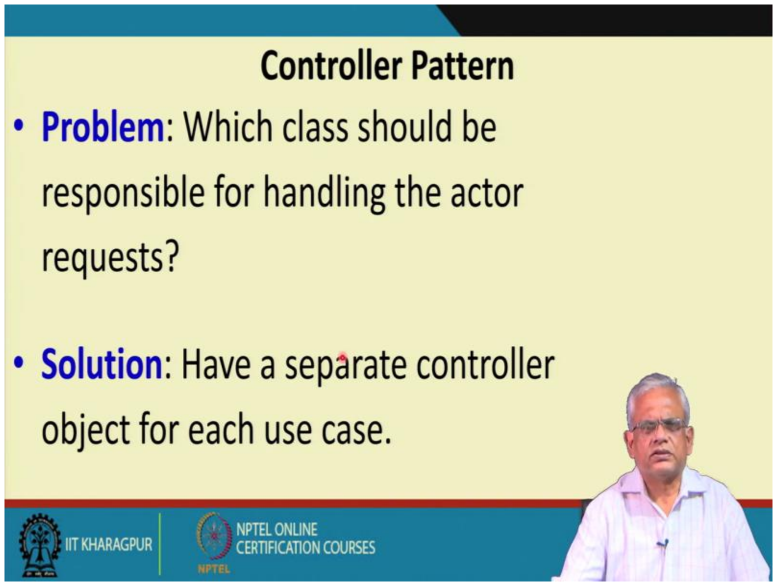
- Creator ensures that coupling due to object instantiation occurs only between closely related classes:
  - An aggregate or container of a class is already coupled with that class
  - Therefore, assigning the creation responsibility to the container or aggregate does not worsen the coupling in the design.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

Now let us conclude our discussion on the create, the creator pattern. Here, the creator pattern ensures that the coupling is the minimum that it can be and the responsibility is given such that it is either given to the aggregator or the composite or to an object, which already has an association with the object.

So the aggregator container has already coupled to that class. So due to the creation, it does not become worse. Assigning the creation responsibility to the container aggregate is appropriate in almost every solution that we work out. We will follow this, that the object creation responsibility lies with the container or the aggregate and this results in lower coupling and better design.

(Refer Slide Time: 6:14)



Controller Pattern

- **Problem**: Which class should be responsible for handling the actor requests?
- **Solution**: Have a separate controller object for each use case.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

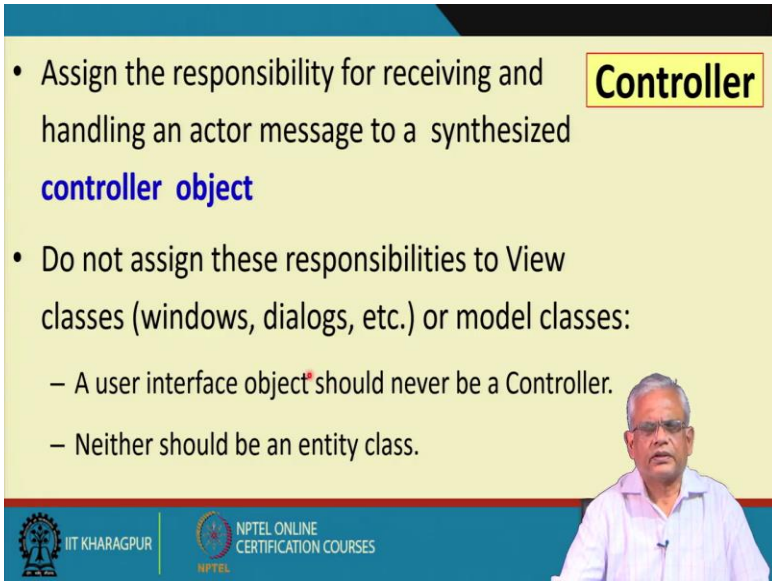
Now let us look at the controller pattern in the general solutions that we have discussed so far, we intuitively use the controller pattern, but then which is based on the idea provided in the GRASP pattern the specific pattern name is controller. The problem that the controller tries to address is which class would be responsible for handling the actor requests. When a use case is invoked by your actor, then a set of actions by various object take place.

But then there has to be a coordinator who orchestrates this action. Once the response comes or the request comes from the user through the user interface, there has to be an object, which knows, which methods of which classes in what sequence need to be invoked. And this sequence is different for different user requests. The solution provided by this pattern is that we have to have a separate controller object for each use case invocation.

The controller objects are short lived. The controller object is created as soon as request from the user comes, it orchestrates the actions of other objects and at the end of the use case, the controller object is the synthetic object. It does not correspond to any physical object, but then as a rule we should have one controller for every use case.

We had been doing that implicitly without really knowing about this pattern. But then this pattern is the origin of that idea, that for every use case, we need to have a controller which knows the business rule for executing that use case. And once the user request comes, the controller object is created and then it does all the necessary actions through requests to other objects.

(Refer Slide Time: 9:04)



Controller

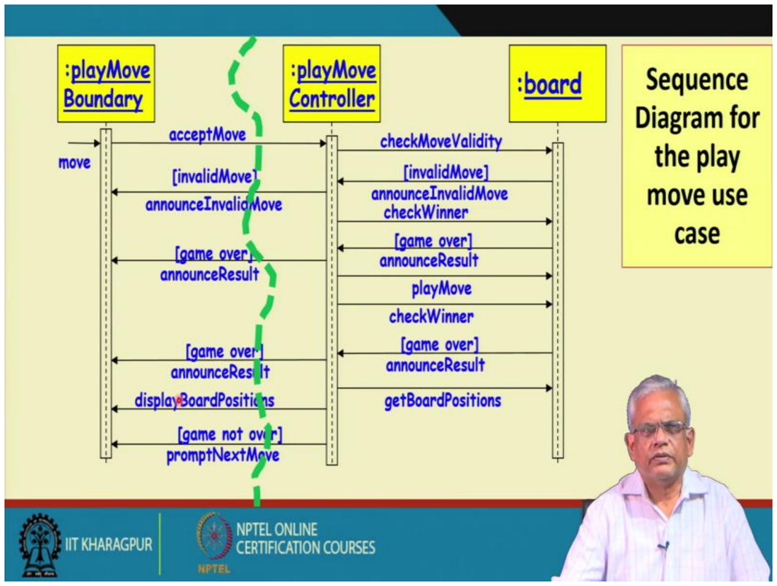
- Assign the responsibility for receiving and handling an actor message to a synthesized **controller object**
- Do not assign these responsibilities to View classes (windows, dialogs, etc.) or model classes:
  - A user interface object should never be a Controller.
  - Neither should be an entity class.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The controller object, is the synthetic object and in response to the actor message it carries out a set of actions by requesting other objects. It is not a good idea not to have a controller. It will really worsen the design, if we assign the responsibilities that the controller used to do to either an entity class or interface class, that would be a bad idea. They will be un-cohesive and also it will become very difficult to maintain.

Neither a user interface object nor the entity object should have the responsibility of the controller. A controller has to be created for every use case. And as soon as there is a user requests, the user interface object handles the user request and requests the controller to coordinate the execution of the use case. And the coordinator takes up, orchestrates the actions of other objects and finally returns the result to the user.

(Refer Slide Time: 10:43)

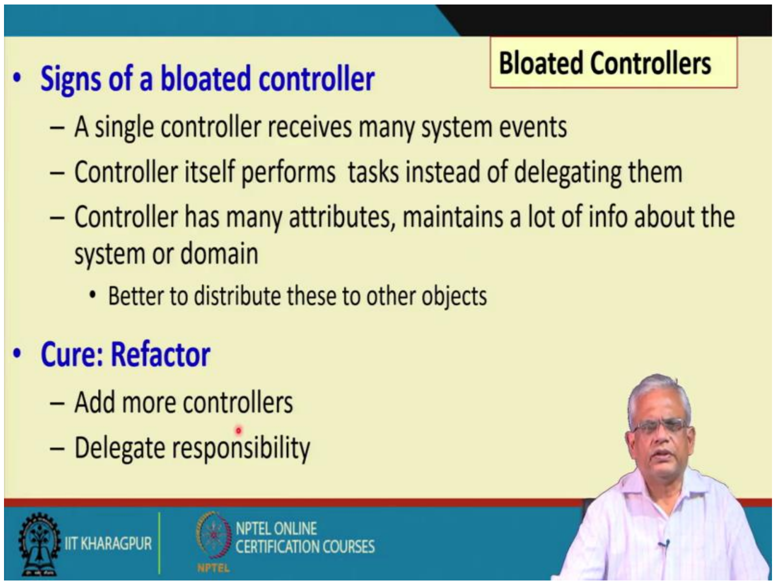

:playMove Boundary
:playMove Controller
:board
Sequence Diagram for the play move use case
move
acceptMove
[invalidMove] announceInvalidMove
[game over] announceResult
[game over] announceResult
displayBoardPositions
[game not over] promptNextMove
checkMoveValidity
[invalidMove] announceInvalidMove
checkWinner
[game over] announceResult
playMove
checkWinner
[game over] announceResult
getBoardPositions
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

We had used this solution for this specific example of the tic-tac-toe. We had only one-use case and one actor, and therefore we had one boundary. And the one-use case, as soon as the use case was invoked on the boundary, then the boundary notified the controller. In this case, the controller object already exists, but in some situations the controller object may get created.

But here the controller object exists is only a very simple problem. And then the controller object knows the business logic and it coordinates the action and first request the board to check the move validity. And then if it is invalid, it announces the invalid move through the boundary. It checks whether the game is won or drawn by any chance, and that it requires the board to carry that out.

And then it requests the board to play the move because that is how the use case works. The play move controller embodies the business logic. And then once the move is played by the computer, then it again requests the board to check the winner. And if there is a winner, our game is drawn, it announces the result. And finally, it gets the board positions and displays the board through the play move boundary.

(Refer Slide Time: 12:52)



Bloated Controllers

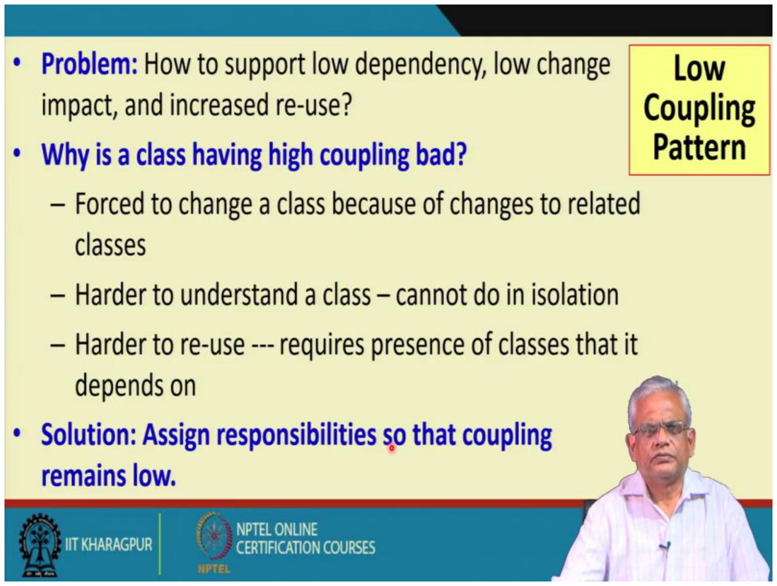
- **Signs of a bloated controller**
  - A single controller receives many system events
  - Controller itself performs tasks instead of delegating them
  - Controller has many attributes, maintains a lot of info about the system or domain
    - Better to distribute these to other objects
- **Cure: Refactor**
  - Add more controllers
  - Delegate responsibility

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The controller objects are very vital part of any solution. They do a lot of activity and especially for complicated use cases, the controllers become very complex. They receive many system events and they perform many actions. Unless we are careful, the controller may become unmaintainable, may contain a lot of bugs and so on. The controller would finally have many attributes, lot of methods and so on. And in these situations, even though at the start of the design, we start with one controller per use case.

But if we find that the controller has become complex, handling many types of events, doing many things by itself, then we need to refactor possibly by adding more controller, give some responsibility of the controller to a secondary controller or if it is doing some specific activities, not exactly the coordinating activities, but then working out certain things, some processing, then maybe it can delegate the responsibility, you can have the control or delegate the responsibility to some entity object.

(Refer Slide Time: 14:39)



Low Coupling Pattern

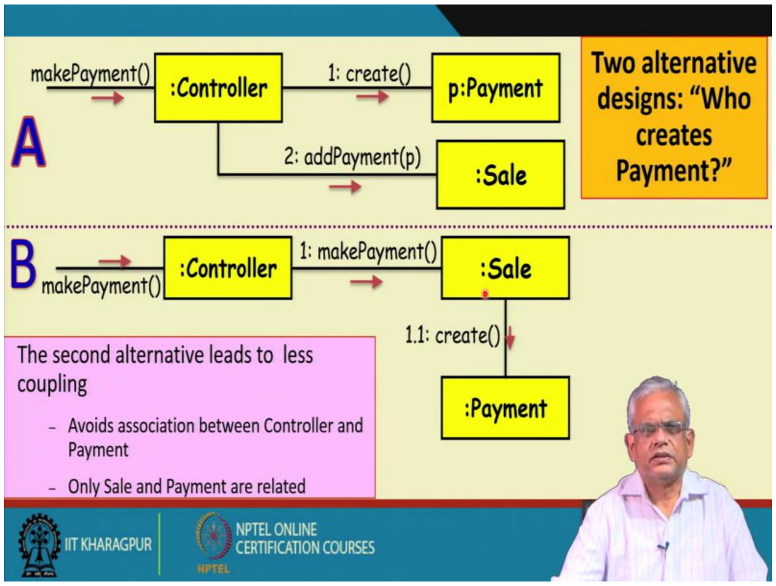
- **Problem:** How to support low dependency, low change impact, and increased re-use?
- **Why is a class having high coupling bad?**
  - Forced to change a class because of changes to related classes
  - Harder to understand a class – cannot do in isolation
  - Harder to re-use --- requires presence of classes that it depends on
- **Solution: Assign responsibilities so that coupling remains low.**

Now let us look at the low coupling pattern. This is more intuitive and more common sense rather than a specific pattern solution. Here the main problem addressed by this pattern is how to support low dependency, low change impact, and increased reuse of a solution. In other words, these problems in a design occur due to coupling between classes, high coupling between classes.

And we know that coupling between classes is bad. It creates many problems, but just to rephrase that if there is a coupling, high coupling between two classes, we cannot just change one class in isolation. If we change one class, we also need to attend the other classes because they are so highly coupled.

And also it is difficult to understand a class in isolation, we need to understand all the coupled classes together that becomes extremely complex. And also reuse scope is very restricted because you cannot just take out a class and reuse. If we want to reuse a class, we have to take all the classes with which it is coupled. But if we find that design is a having high coupling, then this pattern, low coupling pattern suggests that we should reassign the responsibilities so that the coupling remains low.

(Refer Slide Time: 16:34)


Two alternative designs: "Who creates Payment?"

The second alternative leads to less coupling

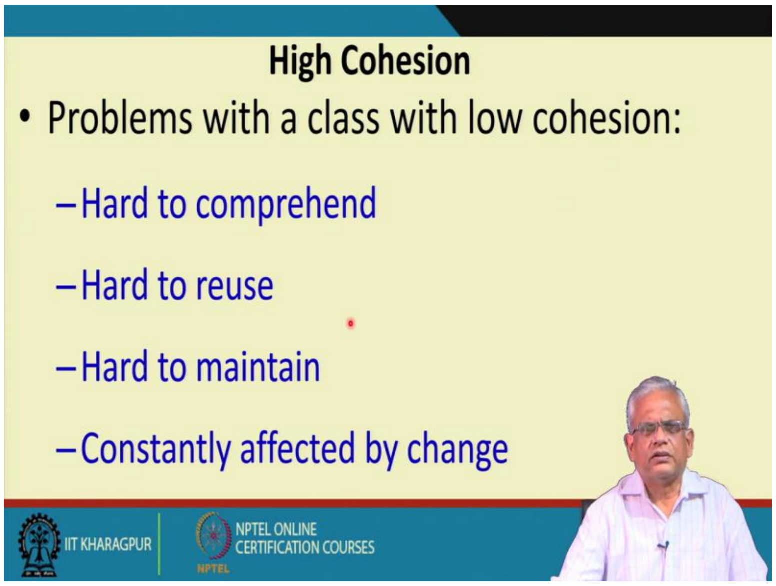
- Avoids association between Controller and Payment
- Only Sale and Payment are related

Just to give an example, the controller here in the figure receives a request of make payment from the user. So this is solution A and this is solution B as divided in the figure, we will see which is better. In the solution A, once the controller receives the make payment request, it receives the payment and creates a payment object. And then it passes on the payment object to the sale with the maternity invocation add payment. And with this, the sale transaction stores the payment object and also the details, the date on which paid and so on in the sale transaction itself.

But an alternate solution here is that the controller on receiving make payment just invokes the make payment on the sale. And it is the sale which creates the payment. And then it stores the idea of the payment and the coupling between the controller and the payment is avoided in this solution B. Of course, it is easy to see that the B solution has lower coupling, the overall design has low coupling.

And this second alternative, the B leads to less coupling, avoids the association within the controller and the payment, and only the sale and payment related. Just notice that we avoided this coupling between controller and payment, by reassigning the responsibilities, instead of controller creating the payment object, it is the sale which created the payment object. Thus tree assign the responsibility and we find that the coupling has reduced.

(Refer Slide Time: 18:52)

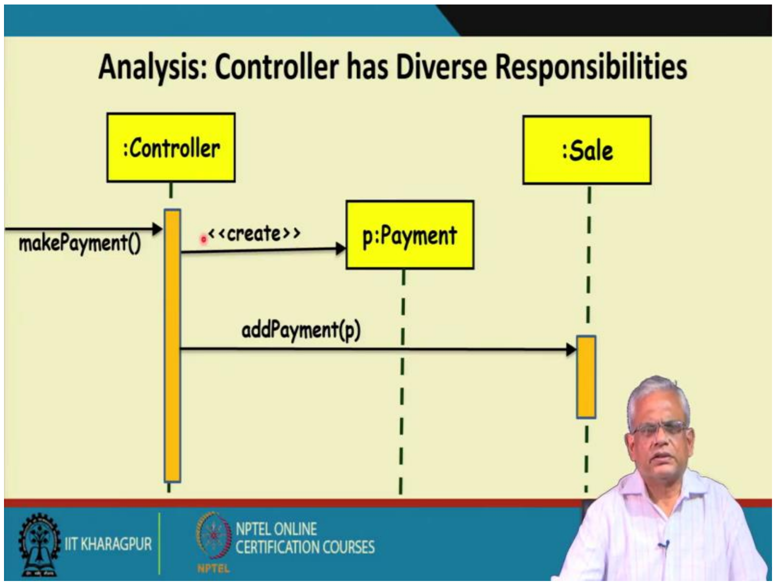


High Cohesion

- Problems with a class with low cohesion:
  - Hard to comprehend
  - Hard to reuse
  - Hard to maintain
  - Constantly affected by change

The next pattern, the next GRASP pattern is the high cohesion pattern. If a design has low cohesion, then there are many problems. It becomes very difficult to understand the design, very hard to reuse anything from this design, very hard to maintain because you change something, some other thing changes and the changes occur very frequently in this type of solution. And the changes affect almost every class. In other words, having a design with high equation is very important and this pattern highlights that aspect.

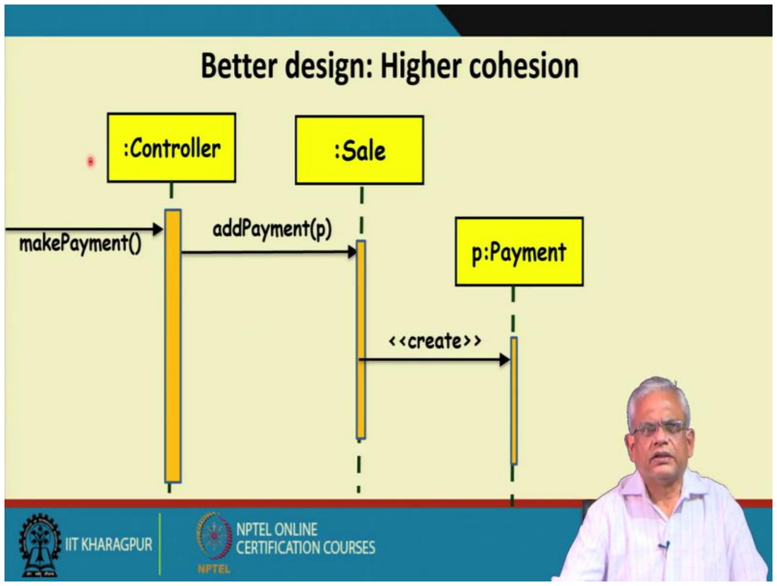
(Refer Slide Time: 19:49)



Analysis: Controller has Diverse Responsibilities

In this example, let us look at the responsibility of the controller. The controller creates payment, and also it coordinates the action of other objects. And therefore the responsibility of the controller is not cohesive. It is not only coordinating other objects, but also it is taking part in some creation and so on. The controller class has less cohesion; we can just redistribute the responsibilities such that the cohesiveness of the controller increases.

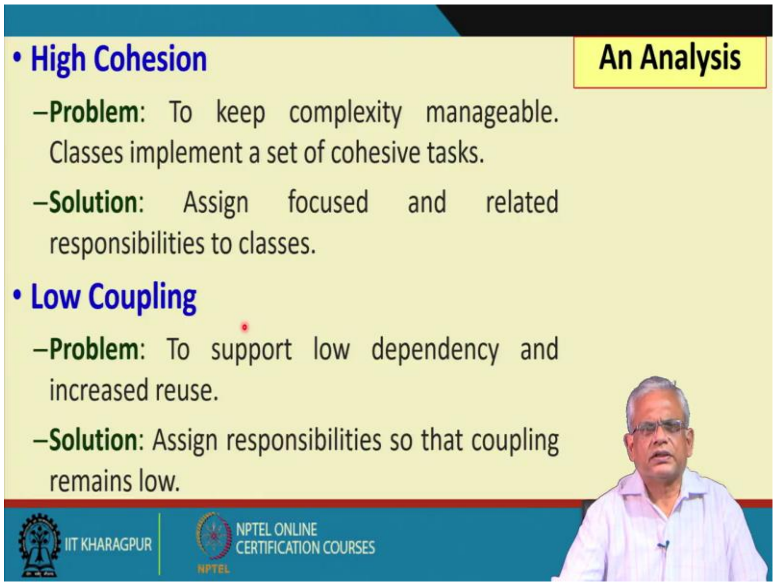
(Refer Slide Time: 20:33)



Better design: Higher cohesion

The controller just coordinates the other objects and then, the P is not there in the makePayment(). This should not be there. It just says controller make sale or something, it invokes here. And then the sale object creates the payment object and stores the ID here, and the controller becomes more cohesive.

(Refer Slide Time: 21:06)

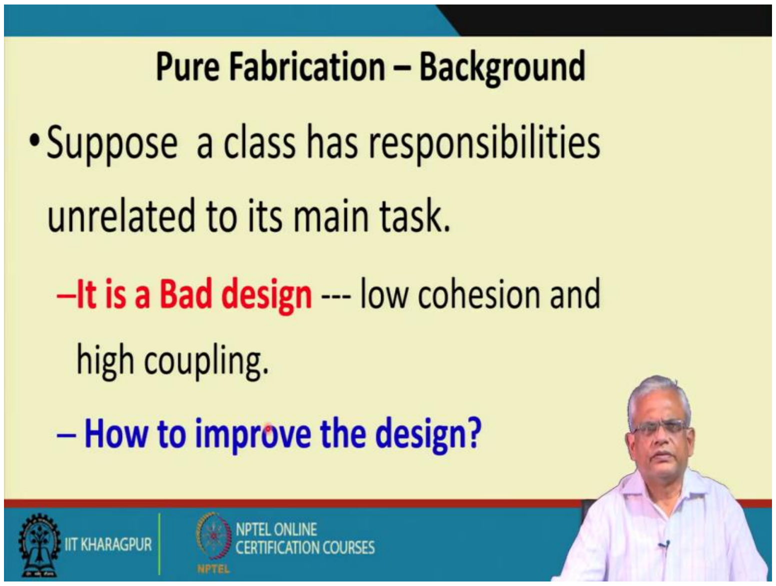

An Analysis

- **High Cohesion**
  - **Problem**: To keep complexity manageable. Classes implement a set of cohesive tasks.
  - **Solution**: Assign focused and related responsibilities to classes.
- **Low Coupling**
  - **Problem**: To support low dependency and increased reuse.
  - **Solution**: Assign responsibilities so that coupling remains low.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

Now we need high cohesion to keep the complexity manageable. And each class performs a set of cohesive tasks. And we so far achieved this by redistributing the responsibility similar was with low coupling. Here to reduce the coupling in a design, we distributed the responsibilities, but now we will look at a pattern where we again focus on reducing the coupling and increasing the cohesion. But in the pattern that we are going to discuss, we do not really redistribute the responsibility, but we create a new object which takes care of the non-cohesive responsibility of your class, so that class becomes more cohesive.

(Refer Slide Time: 22:09)



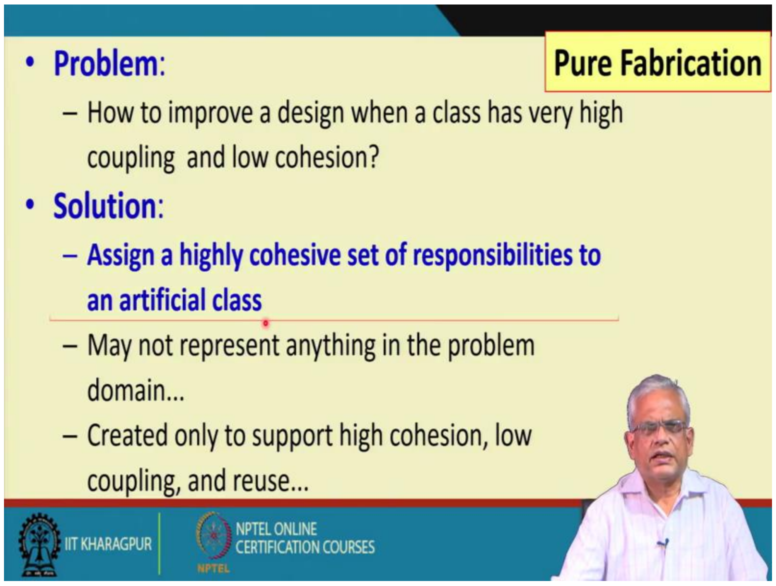
Pure Fabrication – Background

- Suppose a class has responsibilities unrelated to its main task.
  - It is a Bad design --- low cohesion and high coupling.
  - How to improve the design?

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

And the name of the pattern is pure fabrication. When we find that a class is not cohesive, it has responsibility unrelated to its main task. We know that it is a bad design. It leads to low cohesion and also high coupling. And the pattern pure fabrication addresses how to improve the design. Because many times we cannot redistribute the responsibility and increase cohesiveness or reduce coupling because the class to which we assign this responsibility, that classmates will become non-cohesive.

(Refer Slide Time: 22:58)



Pure Fabrication

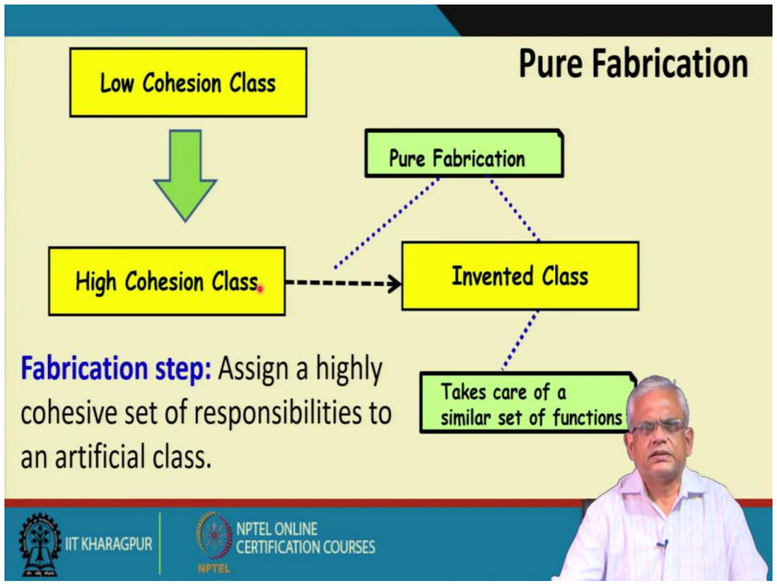
- **Problem:**
  - How to improve a design when a class has very high coupling and low cohesion?
- **Solution:**
  - Assign a highly cohesive set of responsibilities to an artificial class
  - May not represent anything in the problem domain...
  - Created only to support high cohesion, low coupling, and reuse...

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

In these situations, the pure fabrication pattern suggests that we create an artificial class and we assign some responsibilities, such that the non-cohesive becomes smaller cohesive and also the artificial class that we created remains cohesive. This artificial class that we are creating here, need not be identified from the problem domain there may not be anything in the problem domain like that.

But while refining the design, we create these classes through pure fabrication with the idea to reduce the overall coupling and increase the overall cohesion. Again, this is a simple pattern to solve a specific problem of non-coercive classes and high coupling. When the redistribution of responsibilities does not help, we create an artificial class.

(Refer Slide Time: 24:14)

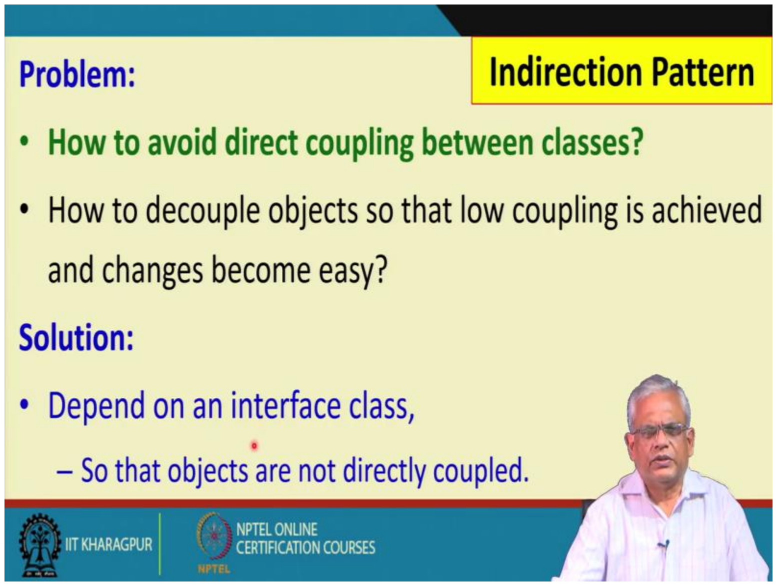


Pure Fabrication

Fabrication step: Assign a highly cohesive set of responsibilities to an artificial class.

If we find a low cohesive class, then we find out what are the methods that are not really the central goal of the class, and we separate them out and assign it to some in winter class we delegate it. And this invented class is the pure fabrication step. And now both the classes which were having low cohesion as well as the inventor class become cohesive. It may be required that we may have more than one inventor class to make the low cohesion class more cohesive, we might need more than one inventor class in certain situations.

(Refer Slide Time: 25:09)



Indirection Pattern

Problem:

- How to avoid direct coupling between classes?
- How to decouple objects so that low coupling is achieved and changes become easy?

Solution:

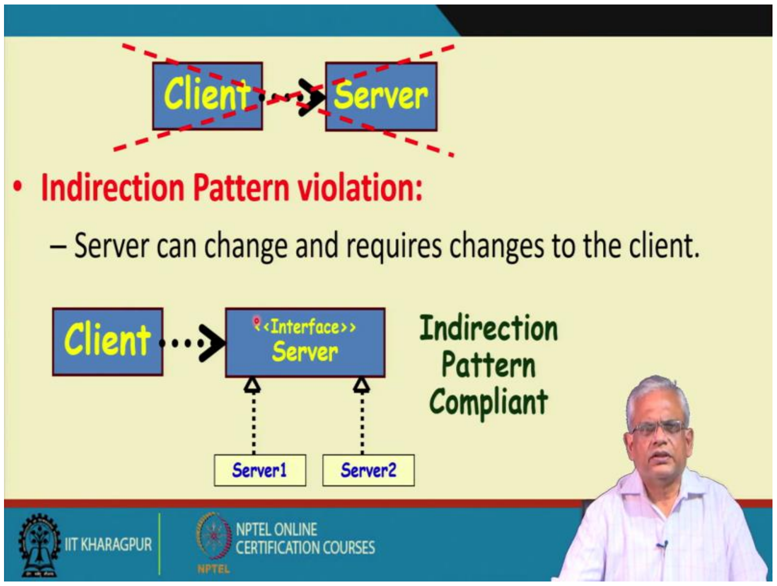
- Depend on an interface class,
  - So that objects are not directly coupled.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now let us look at the next pattern called as the indirection pattern. This pattern again discusses about handling direct coupling between classes. How to avoid coupling? That is the question it addresses. In the overall design, how do we reduce the coupling? The solution proposed here is that many time the coupling cannot be reduced by redistributing responsibility or creating a fabricated class but by using an interface class.

The solution given by the indirection pattern is dependent on an interface, does not depend on a concrete class. Remember that this is one of the basic principle we had discussed before we discussed the patterns and the name of the principle was open-close principle, where we said that high coupling can be avoided by depending on an interface and not a concrete class. And this results in objects which are not directly coupled, the changes do not impact each other.

(Refer Slide Time: 26:40)



- **Indirection Pattern violation:**
  - Server can change and requires changes to the client.

A client object, invoking the service directly on a concrete server object is not a good idea. It is a violation of the indirection pattern. The coupling is bad here because the server can change, any change in the server will affect the client. We should have the server as a interface and different servers, they implement the interface.

In the first design if we change the server object, then the client object will also be changing. And that is not a good idea, violates the indirection pattern. The main reason was that the client and server had become coupled. To break the coupling between the client and the concrete server classes, we use an interface class and the client invokes the methods on the interface class. And this becomes indirection pattern complaint. We are almost at the end of this session, we will stop here. And in the next session, we will complete the GRASP patterns, the remaining GRASP patterns. And we will start discussing about the GoF patterns, the Gang of Four patterns. Thank you.