Object Oriented System Development using UML, Java and Patterns

Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

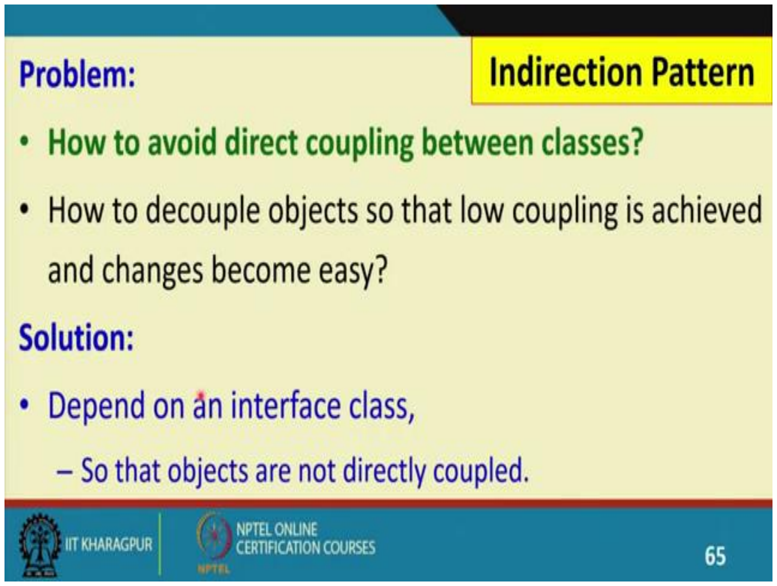
Lecture 41

Introduction to GoF Patterns

Welcome to this session! In the last session we were discussing about the grasped pattern, the indirection pattern, as you might have noticed over the last couple of sessions, we were discussing about the grasp pattern. The central idea is to improve design by using some key ideas. And most of the ideas deal with reducing coupling and increasing cohesion. Towards the end of the last session, we were discussing about the indirection pattern.

The indirection pattern is applicable when the class has bad coupling. We noticed that a class has bad coupling and redistribution of responsibility does not help, neither pure fabrication helps. And we notice here that the coupling is between two concrete classes. As a result if an independent class changes i.e., we have a new version of the independent class, then the dependent class will also have to change. And that is precisely what is addressed by the indirection pattern. The indirection pattern discusses how to avoid coupling between classes.

(Refer Slide Time: 02:17)



Indirection Pattern

Problem:

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

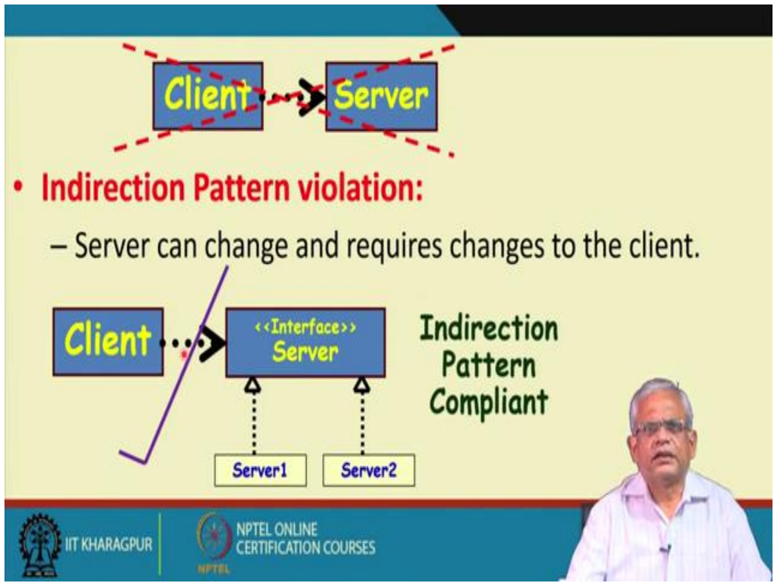
Solution:

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

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
65

And here the coupling is between two concrete objects, and that is what is causing the coupling and the solution given by this pattern is dependent on an interface class and not on the concrete class.

(Refer Slide Time: 02:22)



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

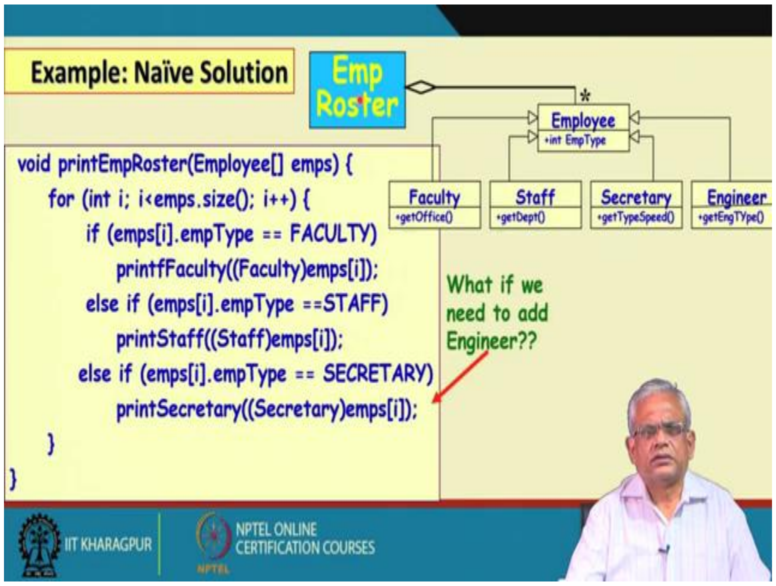
Client
<<Interface>> Server
Server1
Server2
Indirection Pattern Compliant
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The client directly invoking service of a server is not a good idea if in future it may so happened that the server changes. We have new versions of the server, a new type of server. If there is absolutely no chance that the server can change, it is a highly stable and it is not going to change.

Then this solution may be okay that the client directly invokes the server, but if the server class is going to change. We might have new servers and so on. Then this solution is not a good solution. The indirection pattern says that we should not have the client class invoking directly the concrete server class. We should invoke it through an interface, so this solution is indirection pattern compliant. The client invokes the interface.

Methods, and therefore, even if we substitute a server 1 with a server 2 or server 3, as long as they implement the interface, the client will not even need a recompilation. And it insulates the client from changes and reduces coupling.

(Refer Slide Time: 03:56)

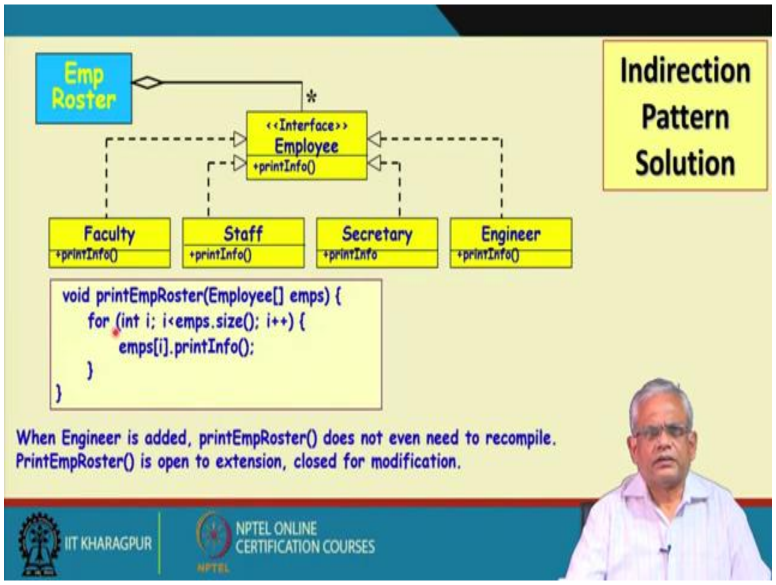

Example: Naïve Solution

```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType ==STAFF)
            printStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
            printSecretary((Secretary)emps[i]);
    }
}
```

What if we need to add Engineer??

Just to give an example, see here the employee roster is couple to a concrete employee class. And the concrete employee class has several derived classes, faculty, staff, secretary etc. Now, since it is a concrete class, the employee roster class, if it wants to print all the employees, then it might have to do something like this. It finds out the number of employees and then for all employees, it finds out the type of employee and then invokes the corresponding methods. And this is not complaint to the indirection pattern. If we add new types of employees, then we need to change the code of the employee roster. It is not a good solution, forces change of the client class on a change of the server class.

(Refer Slide Time: 05:17)



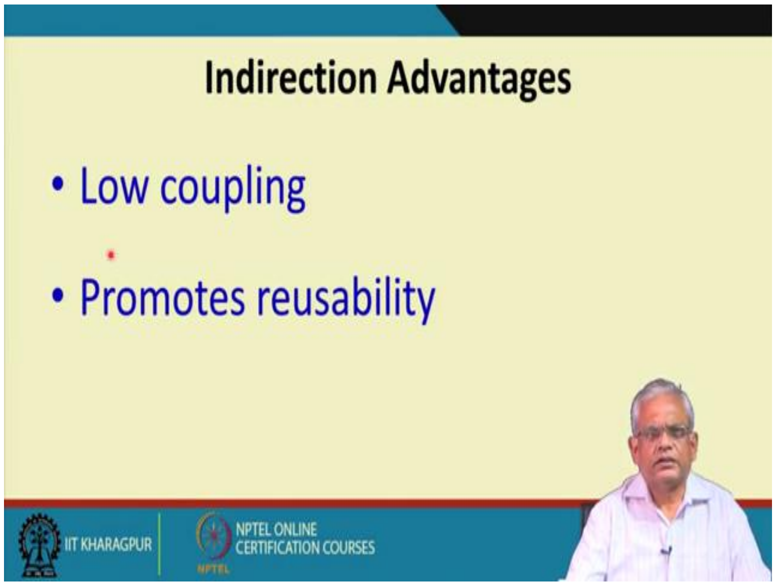
Indirection Pattern Solution

```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        emps[i].printInfo();
    }
}
```

When Engineer is added, printEmpRoster() does not even need to recompile.
PrintEmpRoster() is open to extension, closed for modification.

We should have actually the derived classes implement the employee interface. And the employee roster has an array of objects satisfying the employee interface. And in that case we do not really invokes specific methods and the object, we just invoke the method and the interface. And therefore, we can add more types of objects and then the code of the employee roster does not change.

(Refer Slide Time: 06:04)

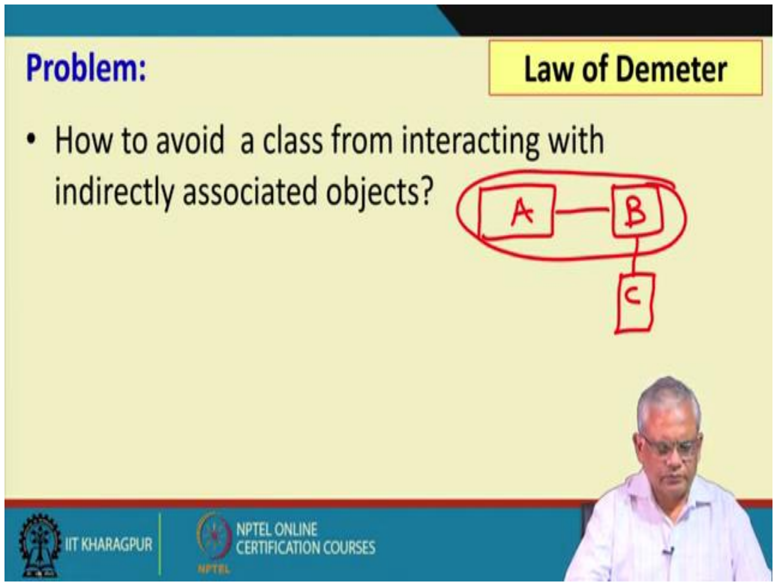

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Indirection Advantages

- Low coupling
- Promotes reusability

The indirection pattern leads to low coupling and also improves reusability because of the low coupling.

(Refer Slide Time: 06:13)



Problem:

Law of Demeter

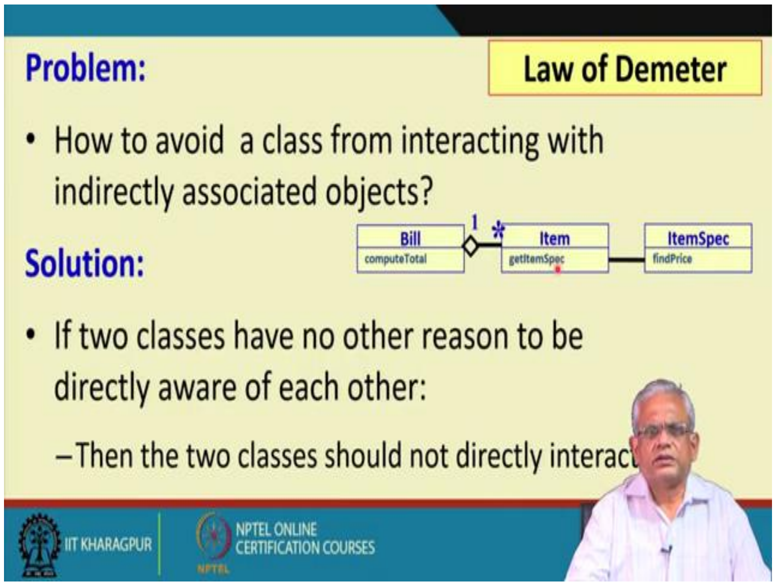
- How to avoid a class from interacting with indirectly associated objects?

Now let us look at the Law of Demeter pattern. The problem that the law of Demeter grasp pattern. It addresses is how to avoid a class from interacting with classes with which it is not directly associated. So, far we had seen that a class can invoke a method of a class with which it is directly associated.

We had a class A which is associated with class B and then class A stores a reference to class B and therefore class A can invoke a method on the B, object B class. But what if we have a C class here which is also associated to B but not directly associated to C. Many times we write the program where we get the reference from B for the C class, A class gets a reference of the C class from the B class and invokes a method on C.

Here, it violates the principle that classes which are closely related or which are associated should invoke methods on each other. Here a class which is not directly associated to another class is invoking the method on C class by getting its ID indirectly. And this is what is the violation of the law of Demeter pattern. Let us see the solution it offers. The problem is identified that a class is trying to invoke method on a class with which it is not directly associated but it is indirectly associated through some other class.

(Refer Slide Time: 08:36)



Problem:

Law of Demeter

- How to avoid a class from interacting with indirectly associated objects?

Solution:

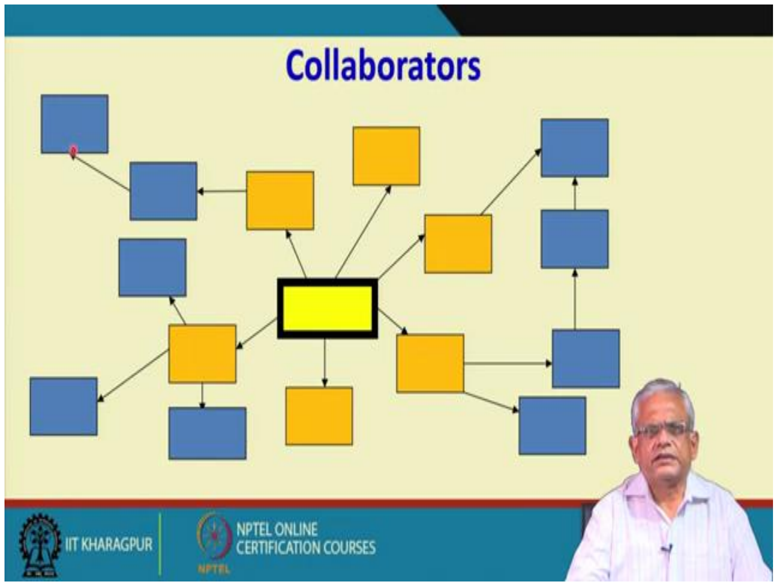
- If two classes have no other reason to be directly aware of each other:
  - Then the two classes should not directly interac

Bill
computeTotal
1
*
Item
getItemSpec
ItemSpec
findPrice
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

This is an example here. Here the bill class has many items, it is aggregation of many items. Which have been purchased, the bill class is an aggregation of many items, and item is associated with specific items specification. The item specification has method like find price. And the item class has method like get item specification. And how many items purchased and so on. Now the bill class has a responsibility to compute the total.

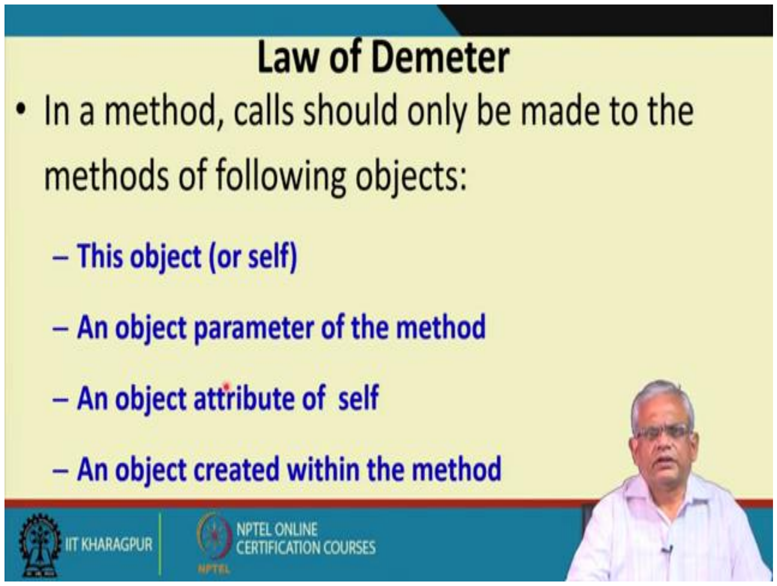
Now, here the bill class is associated with an item class and the item class is associated with item specification when the bill class requests to get the item price. The item has how many numbers purchased and then finds out the price from the item specification and computes the price for the specific items and then it responds to the bill class.

(Refer Slide Time: 10:09)



But what if a class invokes methods on other classes with which it is not associated by getting their IDs from the classes from which it is associated. For example, it may invoke a method on this class here by first requesting this class to get the ID and this class request this to get the ID. The ID comes here and reaches this class and then it invokes a method on this class that will be really bad design and violates the law of Demeter.

(Refer Slide Time: 10:50)



Law of Demeter

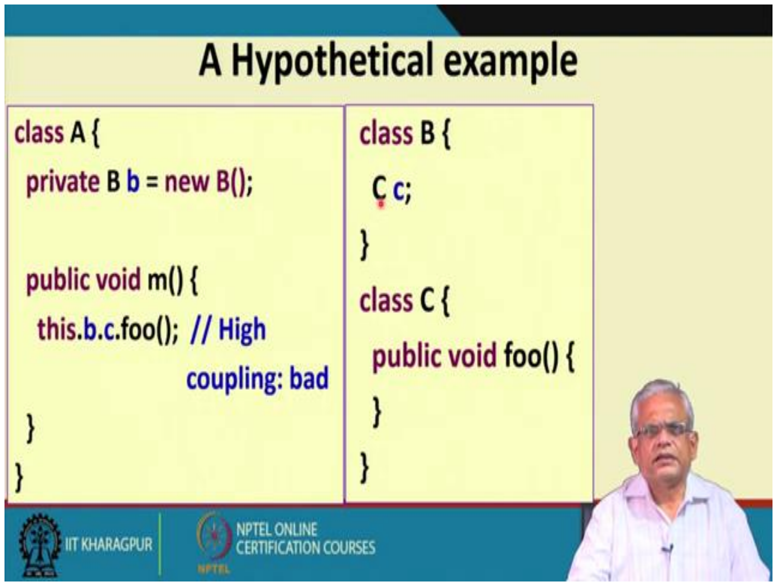
- In a method, calls should only be made to the methods of following objects:
  - This object (or self)
  - An object parameter of the method
  - An object attribute of self
  - An object created within the method

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The law of Demeter says in a method call, the call should be made to an object with any of the following characteristics. Either it should be a self-call that is under this object or it can be a call on an object that is a local variable of this class or an object parameter. It is a parameter pass to the method, then it should invoke the method on the object that is passed as a parameter to the method, or it can invoke a method of an attribute of the class.

An object may be an attribute of a class or maybe an object created within the method and so on. So, these are the cases where the class is directly associated with another class and we are invoking a method on the associated class. And this is alright is compliant with a law of Demeter pattern.

(Refer Slide Time: 12:10)



A Hypothetical example

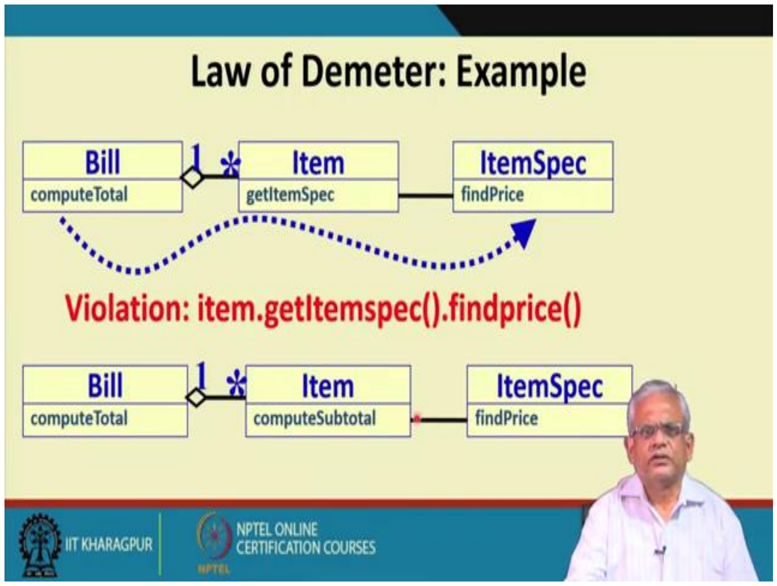
```
class A {
  private B b = new B();

  public void m() {
    this.b.c.foo(); // High
                    coupling: bad
  }
}
```

```
class B {
  C c;
}
class C {
  public void foo() {
  }
}
```

But let us look at this hypothetical example that we have this class A where we have this object reference B an instance of B, and then the method M we are calling this.b.c.foo(). That means the B has the C object visible and we are getting b.c.foo. Foo is the method of the C class and we are invoking in the A class. A method of the C class by calling this.b.c.foo. Class A is not directly associated with C but then in the class A, we are invoking a method of the C class by getting the reference of the C class from the B class. It is a bad design, violates the law of Demeter pattern.

(Refer Slide Time: 13:28)



Law of Demeter: Example

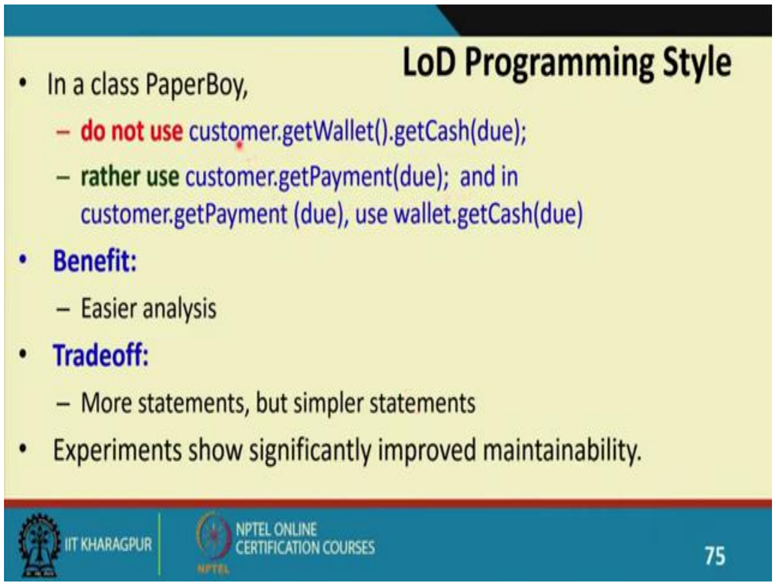
Violation: item.getItemspec().findprice()

Now, same is here, if in this bill, and item, and item spec, if we have this call in the bill class that item.getItemSpec it get the items spec from the bill class. That is the idea of the items spec class is obtained by the bill class by this, and then it calls the find price on the item spec reference. So, it gets the reference of items spec through the item by calling the gate items spec.

And that invokes the find price. It is not a good idea, violates the law Demeter we should have redistributed the responsibilities. The bill class has the responsibility compute total and then it invokes the compute subtotal and the item and the item class computes the price subtotal price by invoking find price on the item spec.

And just observe here that in the law of Demeter we had the problem of a class accessing method of a class with which it is not directly associated. And we solved the problem by redistributing the responsibility so that a class need not have to get the idea of another class and invoke a method on that class, it is simple redistribution of the responsibilities.

(Refer Slide Time: 15:10)



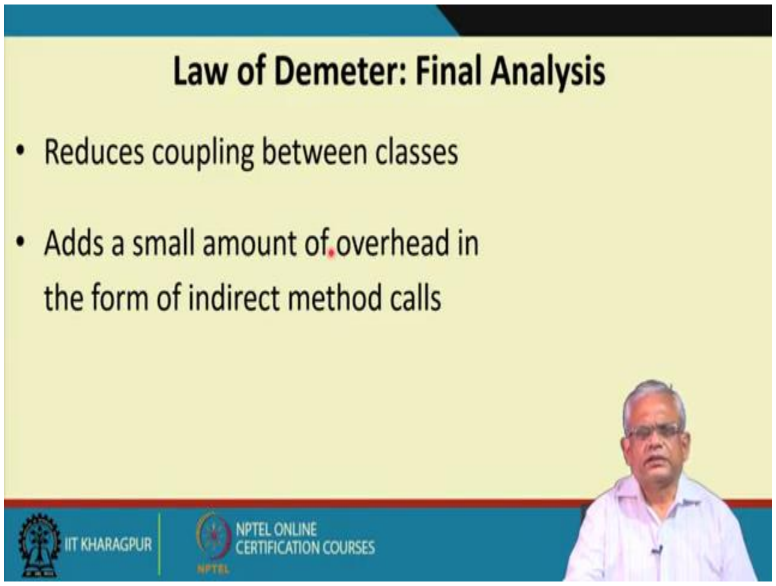
LoD Programming Style

- In a class PaperBoy,
  - do not use customer.getWallet().getCash(due);
  - rather use customer.getPayment(due); and in customer.getPayment (due), use wallet.getCash(due)
- **Benefit:**
  - Easier analysis
- **Tradeoff:**
  - More statements, but simpler statements
- Experiments show significantly improved maintainability.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | 75

The Law of Demeter programming style, we should avoid using customer.getWallet.getCash. We should rather redistribute the responsibility and have customer.getPayment. And in the customer.getPayment, we use the get, we use the wallet.getCash. It reduces coupling, easier analysis, number of statement maybe increase little bit, but then simplifies improves, maintainability.

(Refer Slide Time: 15:57)



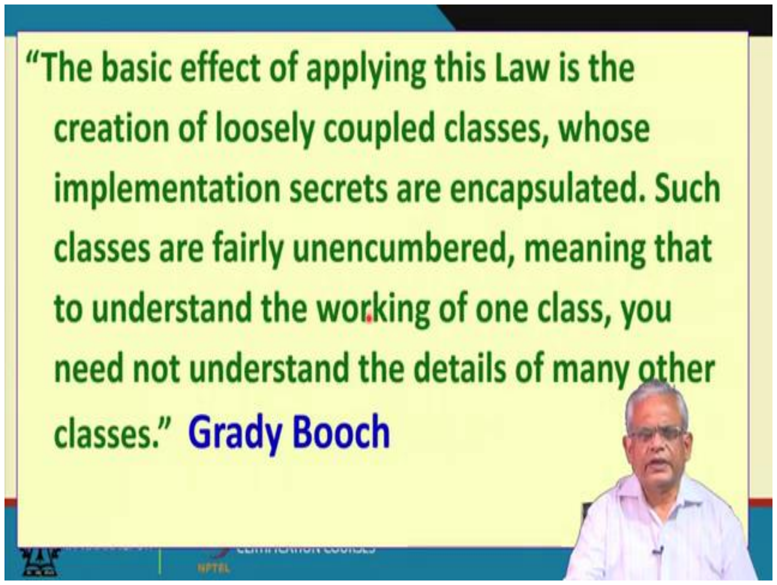
Law of Demeter: Final Analysis

- Reduces coupling between classes
- Adds a small amount of overhead in the form of indirect method calls

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

The advantage of law of Demeter pattern reduces coupling but then if we want to see the negative side of the pattern is maybe number of statements increases and so on. But then we have to avoid by this pattern to get a good design which is maintainable and understandable.

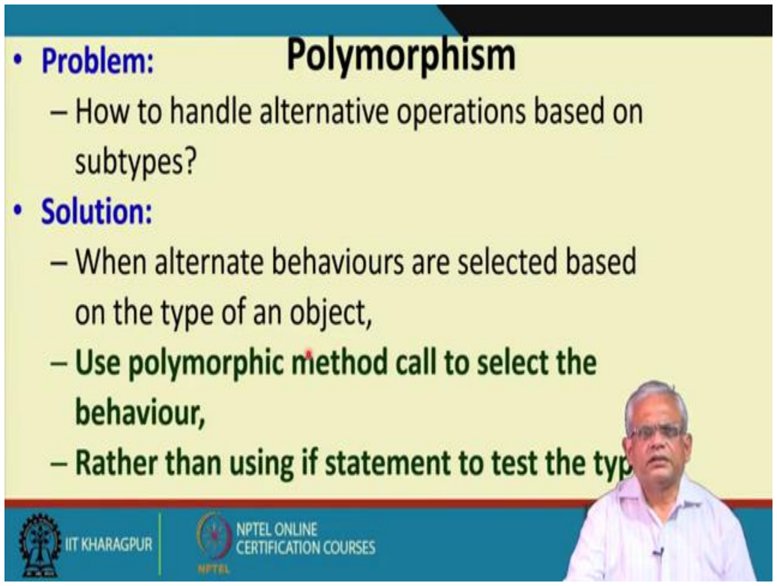
(Refer Slide Time: 16:28)



"The basic effect of applying this Law is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the working of one class, you need not understand the details of many other classes." Grady Booch

In the words of Grady Booch the basic effect of applying the law of Demeter is that we get loosely coupled classes whose implementation secrets are encapsulated. Such classes are fairly, unencumbered meaning that to understand the workings of one class, you need not understand the details of many other classes.

(Refer Slide Time: 16:56)



Polymorphism

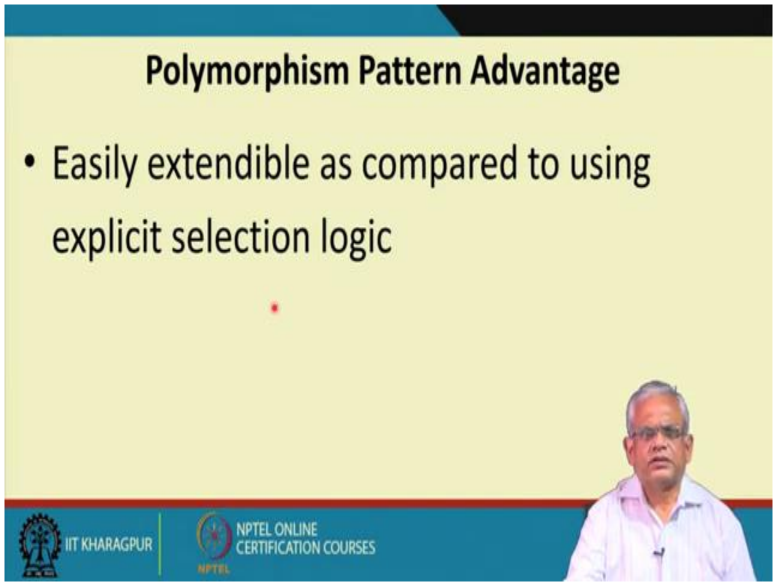
- **Problem:**
  - How to handle alternative operations based on subtypes?
- **Solution:**
  - When alternate behaviours are selected based on the type of an object,
  - **Use polymorphic method call to select the behaviour,**
  - **Rather than using if statement to test the typ**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The last grasp pattern that we are discussing is the polymorphism pattern. This is the 9th grasp pattern again, very intuitive pattern, simple idea, but highly useful. We already know about this pattern from our programming. It handles the problem, that how to handle alternative operations based on subtypes.

The subtypes have different operations. Compared to the base class, how do we handle this and here we use polymorphic method call in the polymorphism pattern, we do not find the subtype and invoke the methods, different methods, we use a polymorphic method call rather than a switch statement based on the type and call the specific methods of different objects.

(Refer Slide Time: 18:04)



Polymorphism Pattern Advantage

- Easily extendible as compared to using explicit selection logic

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

We have already seen several polymorphism examples we will not spend time discussing this pattern. It is very intuitive general-purpose, but this is one of the grasp pattern. The advantage of this pattern, the solution is easily extendable and we get a better solution.

(Refer Slide Time: 18:35)

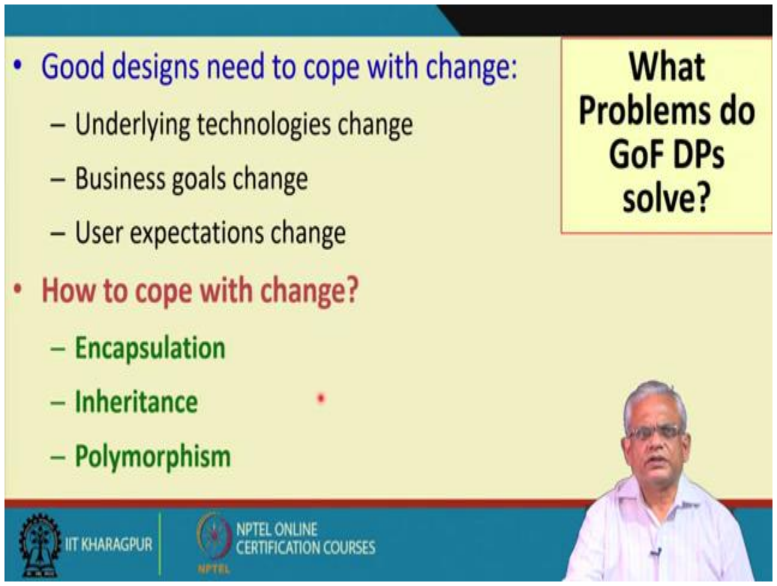
GoF Design Patterns

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now let us look at the GoF patterns. The grasped patterns are general-purpose applicable in almost every design. We do all the 9 grasp patterns are applicable. Even though the patterns are slightly based on intuition rather than a specific class diagram solution or specific code. We will look at the GoF patterns, these are possibly not as widely applicable as the grasp patterns, these are applicable on specific situations.

But then these are more concrete patterns where we can describe the pattern in terms of specific class diagram and when we have an opportunity to use this pattern solution, the GoF pattern solution, it will tremendously improve the quality of the design. But then every GoF pattern we may not have opportunity to use in every design that we attempt to solve.

(Refer Slide Time: 19:56)



What Problems do GoF DPs solve?

- Good designs need to cope with change:
  - Underlying technologies change
  - Business goals change
  - User expectations change
- How to cope with change?
  - Encapsulation
  - Inheritance
  - Polymorphism

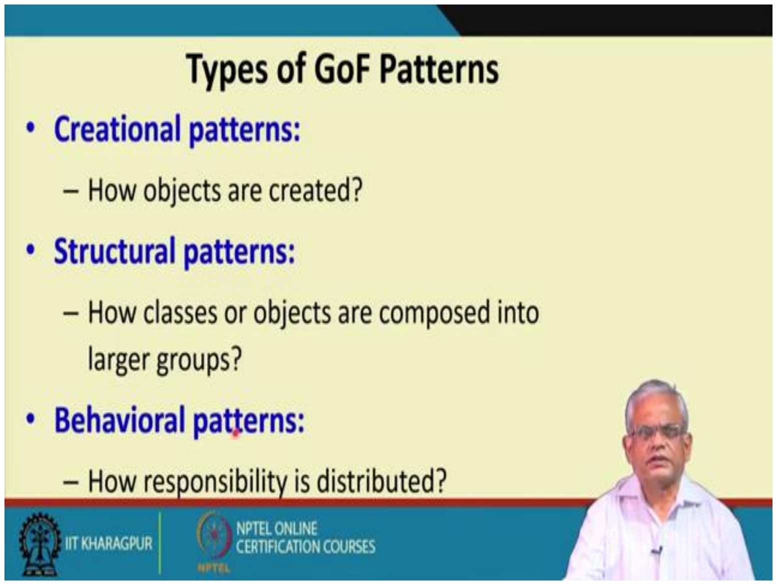
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

First let us understand what are the problems that the GoF design patterns are trying to solve. We can answer this question by saying that the GoF patterns allow a design to cope up with change, changes occur in almost every software solution, changes are universal to a software. There are many-many reasons why software changes maybe the hardware changes, the underlying technologies change, the business goals may change and we need to change the software. Maybe the user expectations change.

Once the user get used to software, they think of new ways or new features which would help them. There are many other reasons also. Why a software changes, but change is universal, change of software, unless we have a good design. We will be overwhelmed by the changes. It will become extremely complex to maintain, it will be extremely costly. And also it may create lot of bugs. The main ways in which the GoF patterns solve this problem is that they use encapsulation, inheritance and polymorphism.

If we give a very high level answer to the question that how do the GoF pattern solve the problem of getting improved maintainability. In the face of a change is that the design solutions are based on encapsulation, inheritance and polymorphism.

(Refer Slide Time: 22:06)



Types of GoF Patterns

- **Creational patterns:**
  - How objects are created?
- **Structural patterns:**
  - How classes or objects are composed into larger groups?
- **Behavioral patterns:**
  - How responsibility is distributed?

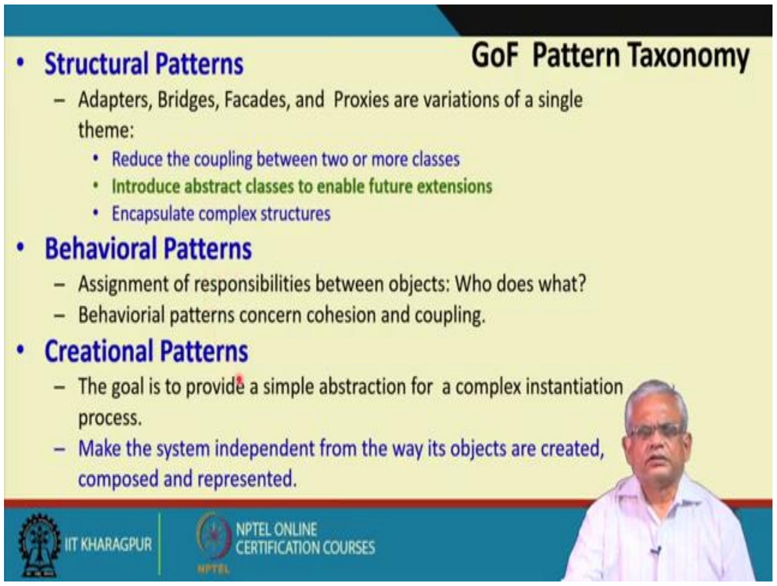
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now let us look at the specific patterns. There are 23 patterns but broadly we can classify the 23 patterns into creational patterns, structural patterns and behavioural patterns. The creational patterns deal with how can objects get created. What is a good design when we need to create objects? We will see there are various types of objects, various situations that we might have to create, and there are several creational patterns in the GoF patterns. The second category of GoF patterns is the structural patterns, you are try to address the question that how to get a larger object, how to compose a set of objects into a larger object.

We will see that there are several structural patterns and they provide very elegant solutions in getting these composite objects. The third category number of patterns exist under the behavioural patterns. Here the question that is addressed is that what is the best way to assign responsibility to classes under specific situations. In the grasp patterns we had also seen the patterns were concerned about how the responsibility can be distributed.

But those were rather general solution here the GoF pattern, GoF behavioural patterns they address specific problems, we will have a look at that as we discuss over the next couple of sessions.

(Refer Slide Time: 23:59)



GoF Pattern Taxonomy

- **Structural Patterns**
  - Adapters, Bridges, Facades, and Proxies are variations of a single theme:
    - Reduce the coupling between two or more classes
    - Introduce abstract classes to enable future extensions
    - Encapsulate complex structures
- **Behavioral Patterns**
  - Assignment of responsibilities between objects: Who does what?
  - Behaviorial patterns concern cohesion and coupling.
- **Creational Patterns**
  - The goal is to provide a simple abstraction for a complex instantiation process.
  - Make the system independent from the way its objects are created, composed and represented.

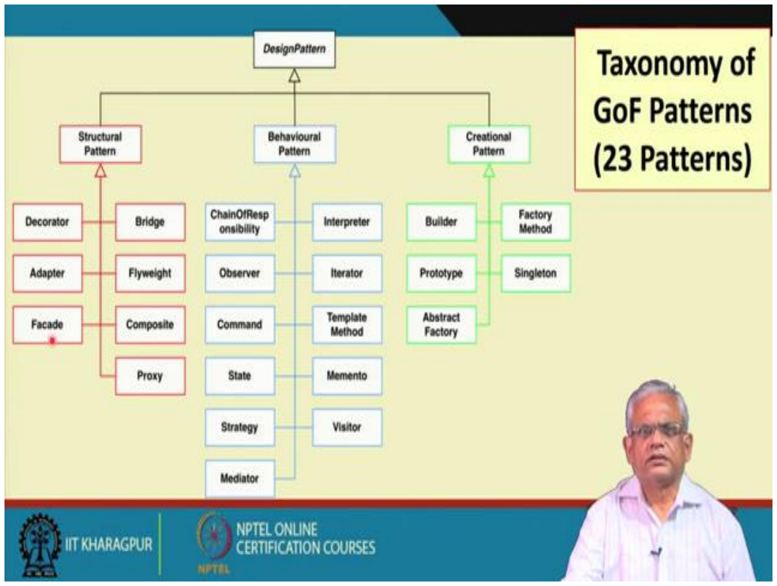
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

We look at few structural patterns, the adapter, bridge, facades and proxy. These are the structural patterns we will discuss and these address the main problem that is how to reduce the coupling between two classes. And we will see in all of these that we reduce the coupling in these patterns of façade, bridge, adapter, proxy by creating abstract classes which help us to in the future extension.

And also we encapsulate complex objects. Those are the key ideas of the structural pattern. And then we have the behavioural patterns, how to assign responsibilities to different objects. And these patterns, again, deal with cohesion and coupling, increased cohesion and lower coupling that is how that is what is achieved by the behavioural patterns.

And the third category are the creational patterns. Here the goal is to provide a simple abstraction of a complex instantiation process. Sometimes instantiating classes is complex and here will see specific solutions about how to create those objects. We will make the system independent of the way the objects are created that is how are the patterns. The factory pattern, the abstract factory, factory method and so on these are some of the patterns that creational patterns.

(Refer Slide Time: 26:02)


Taxonomy of GoF Patterns (23 Patterns)

The 23 patterns as we can roughly classify them as structural patterns, behavioural patterns and creational patterns. The structural patterns, how the objects are structured. Decorator, bridge, adapter, flyweight, façade, composite and proxy these are the ones we will examine. The behavioural patterns, the number of them chain of responsibility, interpreter, observer iterator, common template method, state, memento, strategy, visitor and mediator these are some of the patterns that we will discuss.

And creational patterns, the builder pattern, factory method, prototype, singleton and abstract factory these are the creational patterns we will discuss. We are almost at the end of this session and in the next session we will start discussing the simplest among these, the façade pattern. We will stop here and continue in the next session. Thank you.