Object - Oriented System Development Using UML, Java and Patterns

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

Lecture 34

Open / Closed Principle

Welcome to this session! So far we have discussed about the basic design expertise with the discussions that we so far had given a problem, we can develop a reasonable design solution. But then how to come up with a better design, those aspects we will focus now.

We will first discuss about some principles, which we must be aware to avoid some mistakes which make the code very bad and we should consistently try to make use of these principles wherever possible and we will see that the design patterns which we discussed just couple of sessions away after a few sessions. We will see that the design patterns they embody all these principles, these good principles and if we know this principles, we will be able to understand, why the design patterns do certain things in some way.

For example, we might see that the design patterns make use of the interface classes and the abstract classes very profusely and if we know the design principles we will understand that why they did like that. Otherwise we might consider the design patterns to be unnecessarily complex. To get a good understanding of the design patterns, we must first discuss the basic principles behind good design and that we will do in the next couple of sessions. Let us start discussing about the Object-Oriented Design Principles.

(Refer Slide Time: 02:38)

Symptoms of Rotten Code

- **The system is rigid:**
  - Hard to change anything because for making one change, need to change everything.
- **The system is fragile:**
  - Changes cause the system to break at unexpected places.
- **The system is immobile:**
  - Not reusable.
- **The system is viscous:**
  - The system is opaque —hard to understand.
  - The system is needlessly complex.
  - The system contains needless repetition.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | 48

A bad design, the system becomes rigid and everything is Tangled, you change one thing, the other thing you see that it has to change. The system becomes fragile; you are trying to fix one feature. But as you fix that the other feature has stopped working. The system is immobile and becomes hard to reuse anything from the solution that is developed, because the code is so interdependent that you cannot just take out one part and reuse at other places you have to take everything. The system is viscous that is hard to understand very complex. It is more complex than what it should be lot of repetition. But then it is hard to eliminate them.

(Refer Slide Time: 03:48)

Main Culprit: Dependencies in Design

- Dependencies can arise among classes, packages, and subsystems:
  - Due to a large number of reasons
- Unless dependencies are kept to a minimum:
  - Design and code become unmaintainable
  - Even small modifications break the system
  - Scope of reuse diminishes
  - Too many bugs and test/debug costs rise

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

And this part of the discussion is based on the work of Robert Martin, his book on Object-Oriented Design. And the main culprit here is the dependencies and all these symptoms of bad design and bad code is the dependencies. The dependencies are very high in a bad design. In a good design, we try to reduce the dependencies, the dependencies arise due to a number of reasons.

We will identify what are the reasons and then we will discuss how to reduce these dependencies. If the dependencies are there, then all the problems come like design and maintainance, you modify small part but other part stop working, it is difficult to reuse, difficult to understand and there are too many bugs, it is hard to fix the bugs and so on.

(Refer Slide Time: 05:15)

Common Causes of Dependency

- Tight coupling
- Overuse of inheritance
  - Ex: Extending functionality by subclassing
- Creating an object by explicitly specifying a class.
- Dependence on specific operations
- Dependence on hardware and software platforms

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

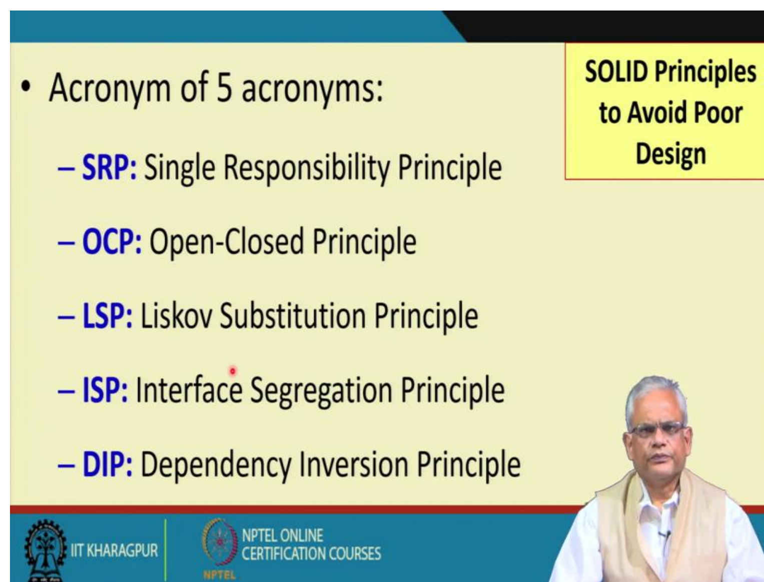
The common causes of dependency, first is tight coupling in our principles that we will discuss, we will identify what gives rise to the tight coupling. Overuse of inheritance. Inheritance is a very important feature in object orientation, it is supposed to help in reducing the effort, make the code elegant, but then many times the developers overuse. We will see the situations where the overuse occurs and that make the design bad. Inheritance, we know that we extend functionality by sub classing, but then this is a powerful tool, but once we have this tool in our hand everything looks like a nail and we try to apply the tool.

Like we have a hammer in our hand and then everything appears like a nail, every fix that we want to do or every development. We want to use the hammer and that will create real problem.

We will identify that what are the problems with the overuse of inheritance and how to avoid them. Creating objects by explicitly specifying the class. We say new and some class, it looks very natural to us the initial coding that we have to use the new method on the specific class and get the object, but we will see that it really makes it increases the coupling.

Dependence on specific operation of concrete classes. This is another problem and dependence on specific platforms. Now, we will start discussing about the principles and we will see that we will try to tackle these problems.

(Refer Slide Time: 07:35)

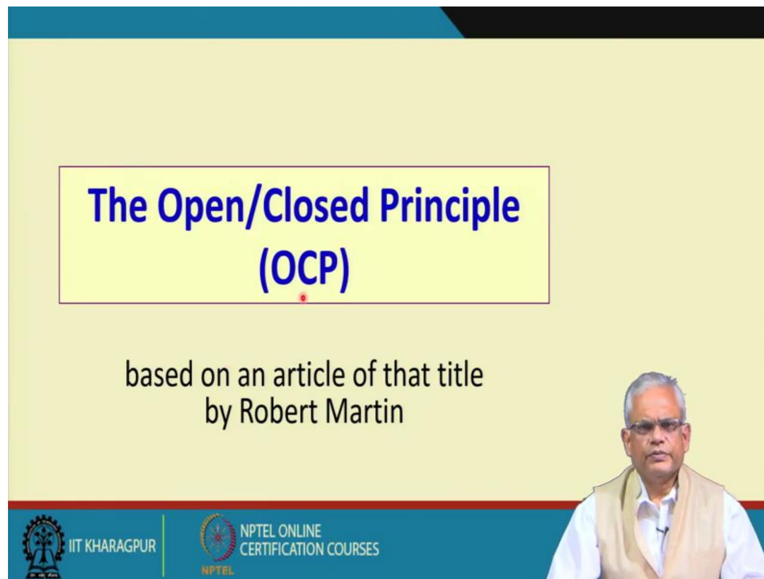


The problems are tackled through five important principles. As mentioned in Robert Martin's book. The first is SRP or the Single Responsibility Principle. OCP or the Open Closed Principle. LSP the Liskov Substitution Principle. ISP the Interface Segregation Principle and DIP the Dependency Inversion Principle. These are five good principles, very important principles to reduce the complexity in the design and reduce the bad things in the design and we will see that the design patterns we discuss are compliant to all these principles.

If we look at the acronym of this 'S O L I D', read these are the five principles and the popularly these five principles are called as the SOLID principles. When you are reading a literature a paper, you might come across these principles. For example, they may mention that violates the Liskov substitution principle or the interface segregation principle and so on and we will try to understand these principles and not only this will help us develop better code, help us in

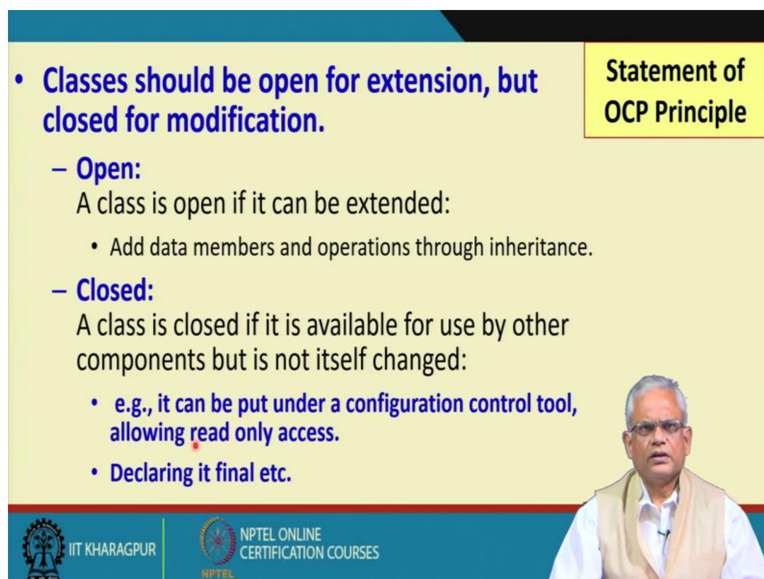
appreciating the design pattern solutions, but also later you will have ample opportunity to come across these principles as your design problems are part of a development team.

(Refer Slide Time: 09:50)



The first principle is based on an article by Robert Martin, which is called as OCP or the Open Closed Principles.

(Refer Slide Time: 10:02)



The statement of this principle is that classes should be open for extension but closed for modification. The main idea here is that once we have developed a class, we have tested it, found it working without bug, then we should not modify it. Because if we modify it, not only that

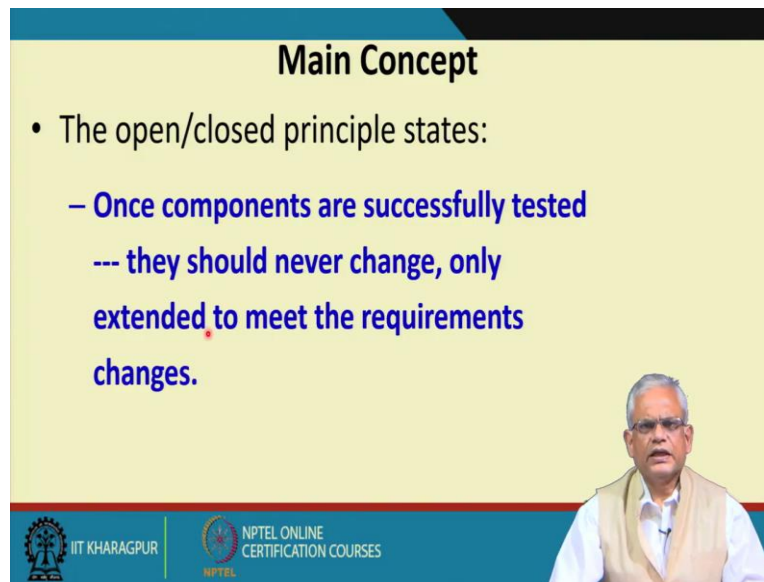
there can be bugs but also the other classes which are making use of the services of this class will be impacted.

Once we have a class, which is working, if we need additional features, we should only extend a class which has been developed and tested. A developed and tested classes should be closed for modification but open for extension. If we need more data members and operations we can add them through inheritance that stands open for extension and closed for modifications is that we should not modify a working code, working-class. The open closed principle, the open closed principle is the principle that whenever we want to add any new functionality or modified functionality, we should only extend the existing class.

If the class is already working and tested there are several classes which invoke its service, if we modify this not only there are chances of bug in an already tested code, but also several other classes will be impacted. They may have unexpected bug because of the changes that we make. So, a working class, we should not change that is closed for change once it is developed and tested. But it should be open for extension, so that is the open closed principle. Once we have a class, which is working tested, we should put under configuration control and will allow only read only access or we may declare final, etc, so that no modification can be done.

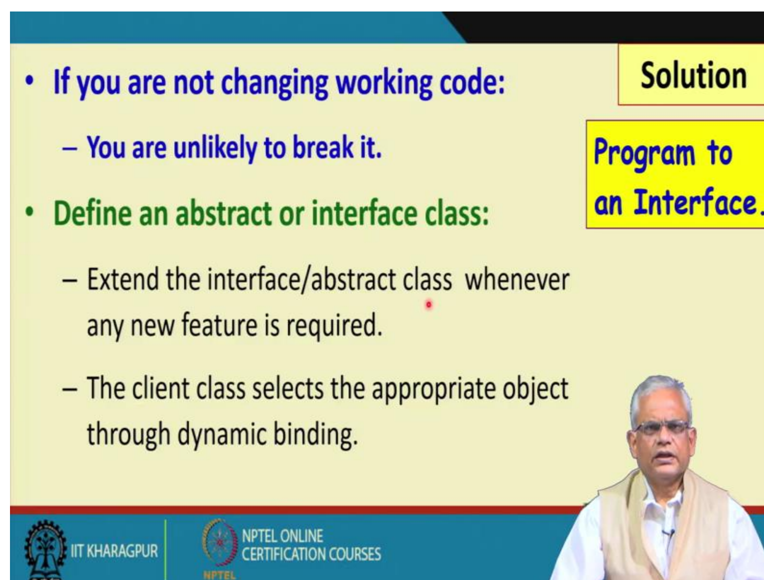
There are various ways we can have the closed implemented. One is that, it will be under configuration control, we only have read access, cannot really change the code. The second is we may declare it final class, etc.

(Refer Slide Time: 13:26)



The main concept of the open closed principle is that once the components are successfully developed and tested we should not change it, but depending on our requirement we might extend it to add additional behaviour to change behaviour and so on.

(Refer Slide Time: 13:50)



The way we implement the open closed principle that we do not change the existing code, so that no new bugs can be introduced in it. But then, we achieve this that to keep something without change is that we use an abstract or interface class. As you know that an abstract class and some of the methods as abstract, then abstract class can have some data members, but some of them

methods are abstract, whereas in an interface class we do not have any data members. We only have the method prototypes.

But then which one to use, an interface class or abstract class. The answer to that question is that if we have some code and data which are not likely to change and reusable across many other classes we will use abstract class. But if the methods are not really common to different classes, each one will have their own different methods, then we will have interface class.

We will use this abstract classes and interface classes. These are the main principle in the open closed principle. Whenever a new feature is required, we will extend the abstract and interface classes and the client classes can select the appropriate object through dynamic binding. This principle is also called as program to an interface.

As long as, we call a method on an interface object or abstract class object we will be implementing the open closed principle. We will just try to make sense, how this is achieved in our next few minutes discussion. The open closed principle is also called as program to an interface, because the solution to the open closed principle is by using an abstract class or an interface class.

(Refer Slide Time: 16:32)

Key Design Idea of OCP

- Open:
  - Can add functionality or data by extending it.
- Close:
  - Can not modify it.
  - Its protocol available for usage by other classes does not change.
  - That is, it has a stable interface
  - Normally implemented with abstract classes

The key idea we had said that open is we can add functionality or data by extending it and close means we cannot modify. Once the class is working it does not change, its interface does not change the other classes which rely on it, they do not have to change because we are not

changing a class which is working and typically the open closed principle is implemented with abstract classes or interface classes.

(Refer Slide Time: 17:15)

The Role of Interface/Abstraction

- It is very difficult to build a class that would never be required to change.
- Two major situations under which changes to a class becomes unavoidable:
  - **Latent errors force change.** We can't fix incorrect operation by extension. The component itself must be fixed.
  - **Performance problems force change.** Met usually by changing a computational algorithm or data structure.
- Changes due to other reasons:
  - Can be achieved through extension.

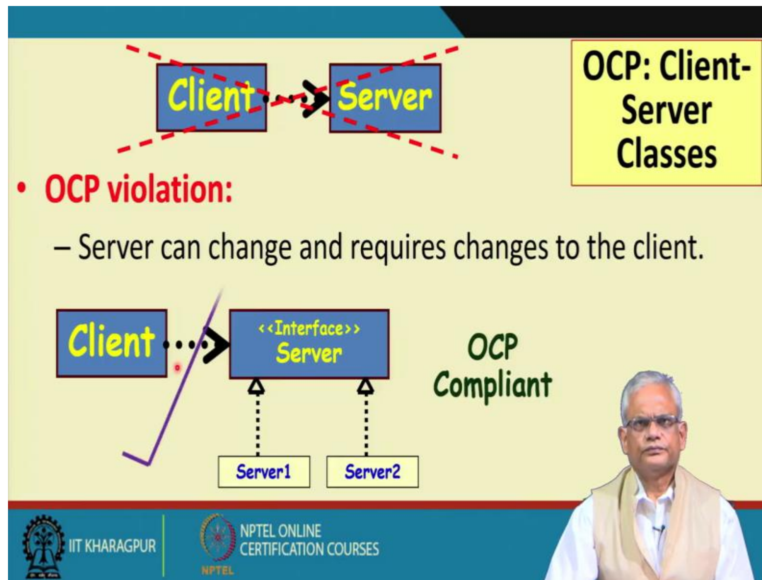
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, let us try to understand the role of the interface and abstract classes here. The statement of the open closed principle that once a class is working. it is close to change, but then to really think of it that a class needs to change mainly on account of two things, one is that if suddenly we find that there is a problem in the class, we would have to change it and also we might say that the performance that it was giving at a later point of time. We might find that this is not satisfactory.

Let us say it was displaying on a terminal. But then we find that the display is too slow over the years and we might need to fix it. If you look at the changes that occur to classes, these are the two main reasons, one is that existing bugs and performance problems and if we need to change it, of course, that is we can always extend it, but then suppose we do not want to just add new features.

But these two problems will not get solved even if we are using extension. One is that the base class itself has problem. In that case, we will have to fix it and also there is problem of the method of the base class we will have to fix it. But then suppose, it is a concrete class. This is always a risk that we will have some errors or there are performance problems. But what about the interface and abstract classes?

(Refer Slide Time: 19:34)



The interface and abstract classes do not have methods actually, only method prototypes and therefore they cannot have bugs latent in them and also there are no performance issues, so we can use the interface classes or abstract classes and make them close to change. If we have this kind of a design like a client is dependent on a server maybe it makes a call on the server, makes a method call then these are dependent, the client is dependent on the server.

It violets the open closed principle, because if the server changes the client need to change. But now, so this is not a good design as it violates the open closed principle, because of the explicit dependency of the client on the server and the server is a concrete class and there are likely reasons for it to change and when it changes the client will get affected.

The bug may propagate or we change the interface and the client code has to be Rewritten. The OCP complaint solution will be that client is dependent on an interface and that same interface can be implemented by several servers. In this case, the client does not get affected even if the certain server has a problem either a performance issue or a bug. The client code does not have to get recompiled and bugs cannot propagate to client code because we are just invoking an interface, the interface itself does not change.

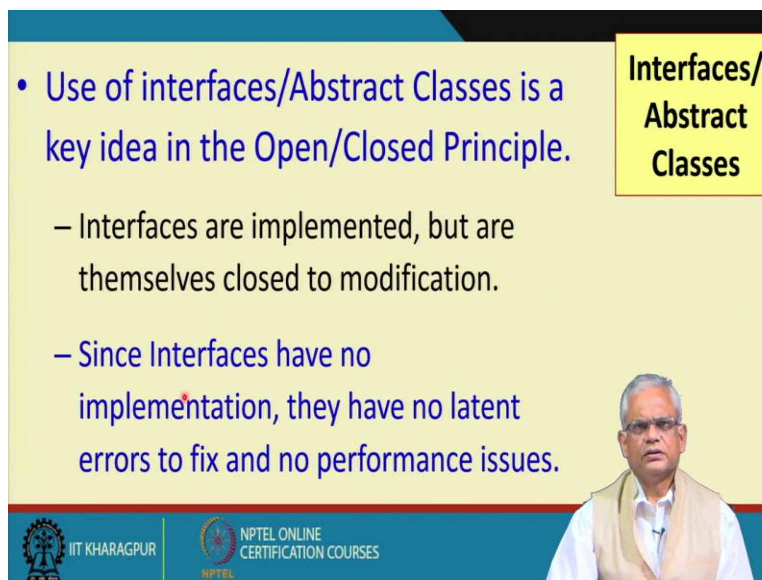
And this is the basic principle of OCP, that instead of a client class directly calling a server class we have the client calling to an interface. This is program to an interface and then there can be several server classes implementing the interface. We can even extend the interface later if we

want some interfaces need to be changed or new services are required. We might extend the interface but this remains the working code does not get affected and this is the solution to the open closed problem is programming to an interface.

But one thing must mention here is that this kind of design is required. If we anticipate that there can be changes to the server class. But suppose we have a situation where the code is already working for many years. It is a concrete server class and the client is invoking directly the service of the concrete sever class and there is no reason for the server class code to change, there is no bugs, no performance issues can arise, in that case possibly this design itself may be okay.

But in all other situations where the server class can change either due to bugs or performance consideration or new features required, we have to use the OCP solution, where we have the concrete classes, server classes implement the interface, server interface and the client invokes the methods on the interface

(Refer Slide Time: 24:13)



The use of interface classes and abstract classes are the key ideas in the open closed principle. We will see that most of the design patterns profusely use the interface classes and abstract classes and one of the reasons they do that is to be complaint with OCP, the interfaces are implemented by many classes but the interface itself is closed for modification.

Interfaces do not have any code they are implementing methods and so on. So, there is no chance of a bug. The interface does not have to change and there are no performance issues as well because the interface class does not have the method implementations and therefore the solution to the open closed principle is to use the interface classes and abstract classes, whenever a client class needs to call a server class and the server class need to change due to various reasons.

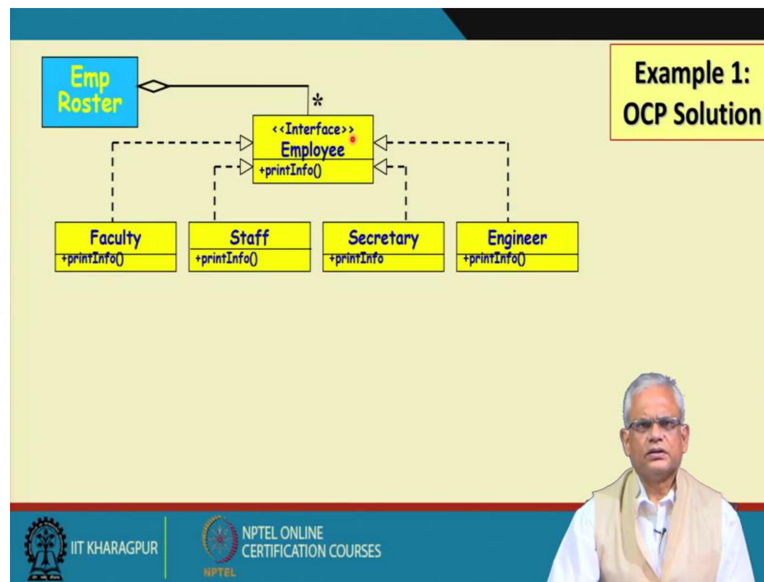
(Refer Slide Time: 25:39)

Now, let us look at an example, violation of the open closed principle and how we can fix it. Let us say we have this solution, where we have this employee roster consists of many employee objects, the employee is a concrete class having employee type, etc. and some basic methods. Now, we have various types of employees. We have faculty, staff, secretary, engineer, etc.

Now, there are several other classes which invoke services of the employee, because employee is a concrete class and these other classes invoke the specific employee classes, but employ itself is a concrete class. The code of the employee in this case will look like this. Because employee is a concrete class.

We have all the employees stored in an array and depending on the employee type we invoke specific methods on it and this is not a OCP complaint code, because the employee is a concrete class. We can make it OCP complaint. So, there are various reasons why the employee may change, the employee class may change because there are performance issues or there are bugs in the employee or there may be new employees added here, and so on.

(Refer Slide Time: 27:38)



The OCP complaint solution is that the employee roster aggregates employees, where employee is an interface and the different concrete classes implement the employee interface and here even though there is a bug in these classes or their performance issues or new employees are added the employee rooster, sorry the classes which are invoking the employee. The methods of the employee do not have to change.

So, this is the main idea here very important principle the open closed principle. Once we discuss design patterns, if we know this principle we will understand that why the design patterns have been designed the way they are with this first principle of open closed principle here out of the SOLID principle, we will discuss one principle, we will discuss the other principles in our subsequent sessions, we will stop here and continue in the next session. Thank you.