Object - Oriented System Development Using UML, Java and Patterns

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

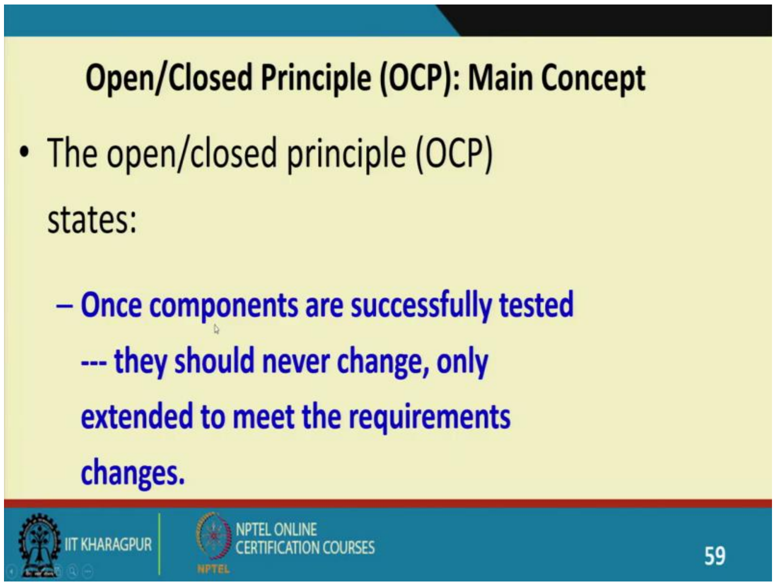
Lecture 35

LSP, ISP Principles

Welcome to this session, over the last few sessions we had looked at the Basic Object-Oriented Design Process using UML. Given a problem description we could arrive at reasonably good design but then we remarked that to arrive at a better design we need to know certain principles otherwise the code might exhibit different types of problems. And one of the causes or culprits in this regard is the dependency between classes.

And we were discussing five important principles which help us avoid these dependencies. The first principle we were discussing was the open closed principle or OCP. Let us just refresh the OCP principle and then we will go on to the other principles and then we will start discussing about the design patterns and see that these principles play a crucial role in the design pattern solutions, let us start.

(Refer Slide Time: 1:48)



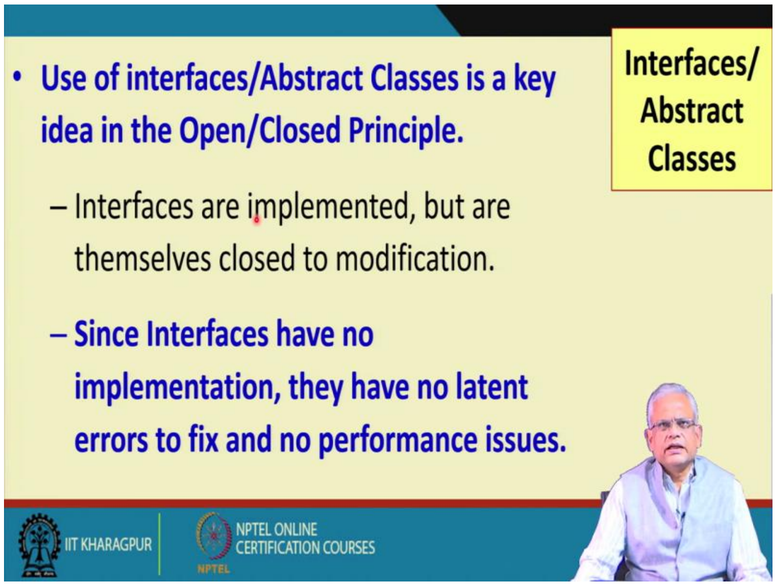
Open/Closed Principle (OCP): Main Concept

- The open/closed principle (OCP) states:
  - Once components are successfully tested --- they should never change, only extended to meet the requirements changes.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
59

The open closed principle, if we remember the main concept here is that a class once has been tested should not change, we have already developed and tested it, it should not change. We should ideally put it under configuration control or read-only access. Whenever we need any changes we should extend the class to meet the requirements.

(Refer Slide Time: 2:32)



Interfaces/ Abstract Classes

- Use of interfaces/Abstract Classes is a key idea in the Open/Closed Principle.
  - Interfaces are implemented, but are themselves closed to modification.
  - Since Interfaces have no implementation, they have no latent errors to fix and no performance issues.

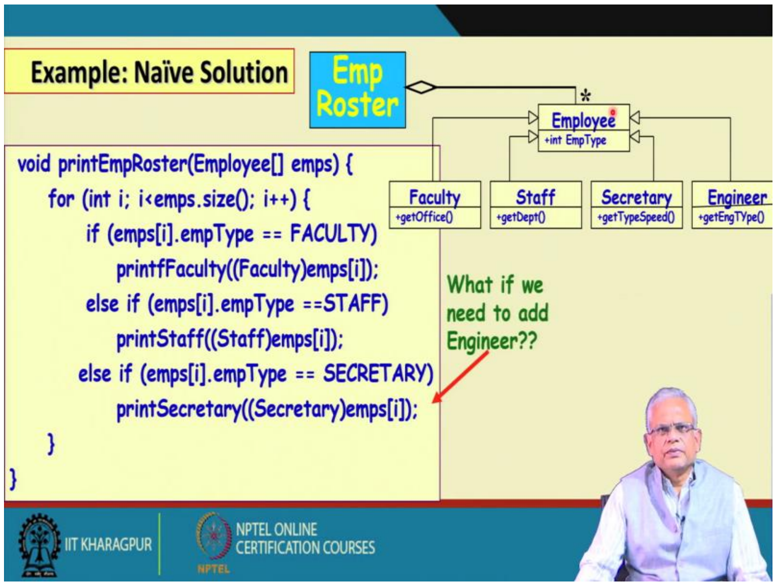
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

But then we said that it is possibly a good idea that once code is working, we should not change it, but then the real codes need change due to various reasons, bug fixes, performance, enhancements and so on. And then we said that the interfaces and abstract classes play a crucial role here.

If we want to enforce open-closed principle, we need to use abstract classes and interfaces. The main reason here is that once the abstract classes are developed, they do not need any change because they do not have performance issues, they do not have code and therefore they do not have bug fix problems.

And therefore once the interfaces have been developed we can put these under configuration control and the other classes can implement the interfaces. But the question that somebody may ask is that what if the method prototypes themselves need change for some applications. In those cases, we will derive new classes from the implementation, from the interface classes to accommodate those changes. And the existing clients will not need any change. If we did not do that the existing code which use the interface also might need change, so these are the central ideas in the open-closed principle.

(Refer Slide Time: 4:50)


Example: Naïve Solution

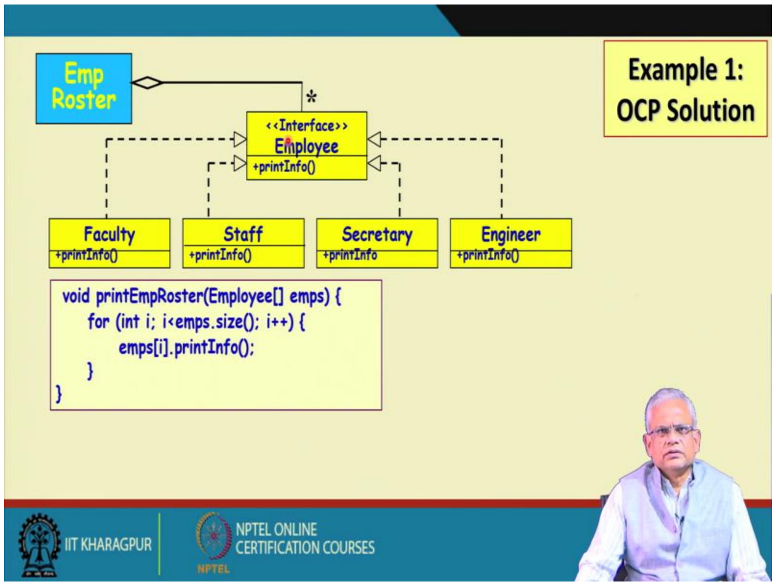
```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType ==STAFF)
            printStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
     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

What if we need to add Engineer??

We were trying to give an example, in the last session we have this employee roster which consists of many types of employees, faculty, staffs, secretary, engineer and so on. And see here that employee is a concrete class and it has a data item ‘employee type’. And the programmer has written the code that based on the employee type identifies the different specific types of employees and calls the appropriate methods there.

For example, print faculty, print staff, print secretary, print engineer, etc. is an example. The main problem with this code is that it violates the open-closed principle. If we need to add another employee type here, we will need to change the employee base class and recompile. Then the question is that how do we make it OCP compliant.

(Refer Slide Time: 6:18)

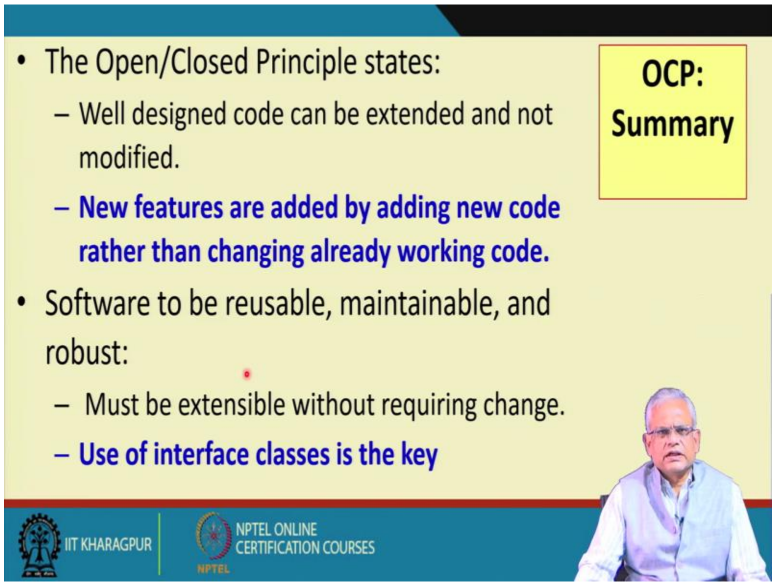


Example 1: OCP Solution

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

To make it OCP compliant, we need to use an interface employee and the specific types of employees they implement the employee interface and therefore the provide definition to the print info for the different employees. And then we can have the code where we can add another type here without disturbing the other classes or the employee roster or the other classes which are clients of the employee, we do not have to make any changes. And this code is OCP compliant because once the employee interface has been developed and tested we can have only read-only access or put it under the configuration control.

(Refer Slide Time: 7:18)



OCP: Summary

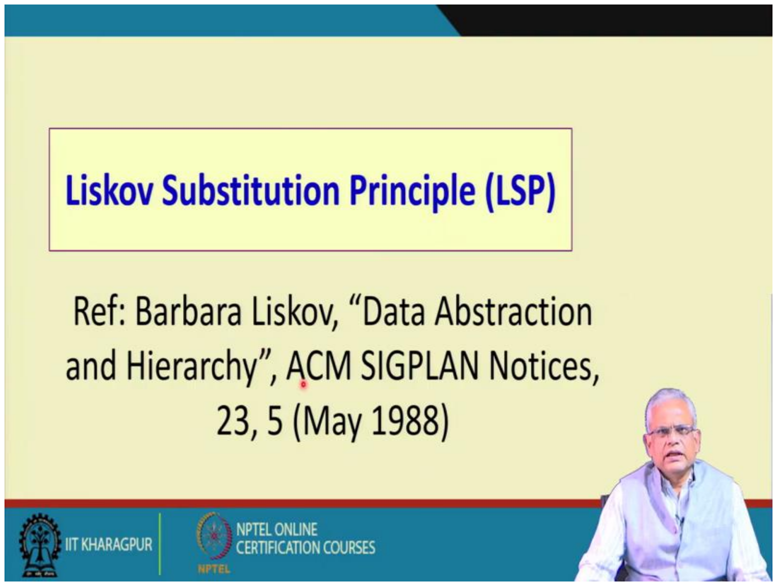
- The Open/Closed Principle states:
  - Well designed code can be extended and not modified.
  - **New features are added by adding new code rather than changing already working code.**
- Software to be reusable, maintainable, and robust:
  - Must be extensible without requiring change.
  - **Use of interface classes is the key**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The summary of this principle is that once a class has been tested, developed and tested, we do not change it, it is closed for change but it is open for extension we can add new features by extending the class rather than changing the already working code. Every software needs maintenance but that we achieve through extension and the main way we achieve OCP is by using the interface classes and the abstract classes.

But let me just again repeat here that when we can use the interface classes and when we can use the abstract classes the answer to this lies in the fact that if the methods in the interface class differ from the concrete classes, the different concrete classes have different implementations of the method defined in the interface class, then we use interface class. But if there are some methods which are shared across all classes that the concrete classes then we use abstract class and we put the reusable method in the abstract class.

(Refer Slide Time: 9:05)



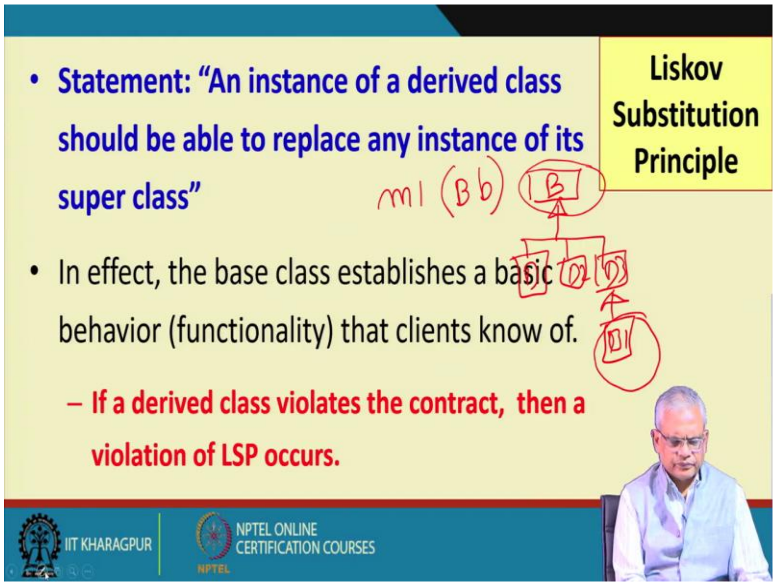
Liskov Substitution Principle (LSP)

Ref: Barbara Liskov, "Data Abstraction and Hierarchy", ACM SIGPLAN Notices, 23, 5 (May 1988)

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

Now, let us look at another principle the Liskov Substitution Principle, this principle was proposed by Barbara Liskov in the ACM Sigplan Notices in May 1988.

(Refer Slide Time: 9:26)



Liskov Substitution Principle

- Statement: "An instance of a derived class should be able to replace any instance of its super class"
- In effect, the base class establishes a basic behavior (functionality) that clients know of.
  - If a derived class violates the contract, then a violation of LSP occurs.

The statement of the principle is that "an instance of a derived class should be able to replace any instance of its super class". So, if we have a base class, and we have several derived classes, D 1, D 2, D 3, D 31, etc., then an instance of D 31 we should be able to use where B is used. For example, in a method call let me just write the method call as m1 and which needs an object of type B.

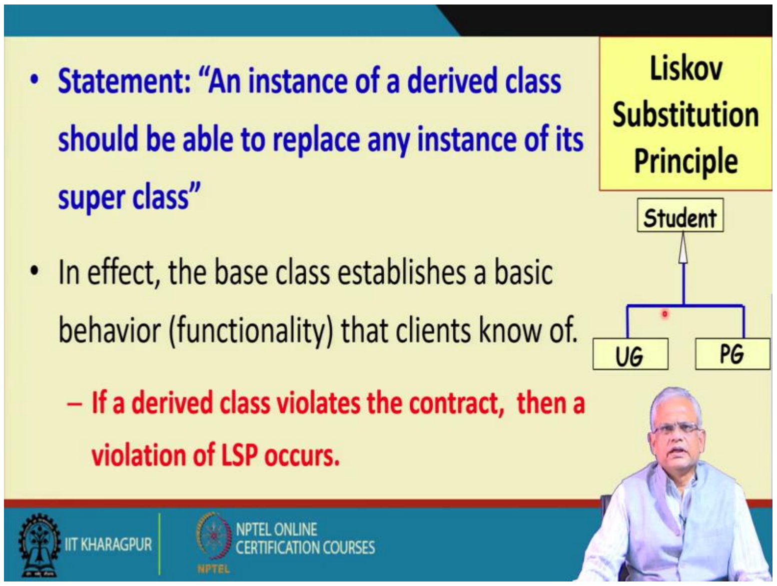
In this case, it should be possible to use an instance of the D31 class or D1 class or D2 any derived classes, we should be able to use in place at the situation where an instance of B is required. That is the implication of an instance of a derived class should be able to replace any instance of its super class seamlessly.

The reason why it is possible is that the base class provides a behavior, the basic behavior which the clients of the B class aware of and since all the derived classes comply to the behavior of the base class, therefore, we can replace an instance of B with any instance of the derived class. But can we replace an instance of a derived class with an instance of a base class? The answer is no, because the behavior that D1 provides is not really the behavior that B provides. Some of the behaviors are same basically the ones that are inherited in D1, but D1 might have other methods implemented which B1 does not support. And therefore, wherever an instance of D31 is expected we cannot use an instance of B.

But just see here that the behavior of D1, D2, D3 and D31 have to comply with the behavior of B. If there is any difference in the behavior of D1 from B then there is a violation of the Liskov substitution principle. That is, we cannot use an instance of D1, D2, D3 etcetera in place of an instance of B.

When the clients use an instance of D1, D2, D3 they will have to do something specific, if there is a violation. If they are compliant with B that is in a typical normal inheritance hierarchy, then the clients of B will not even know that they are dealing with an instance of D1, D2, D3, etc. So, that is the main principle here.

(Refer Slide Time: 13:58)



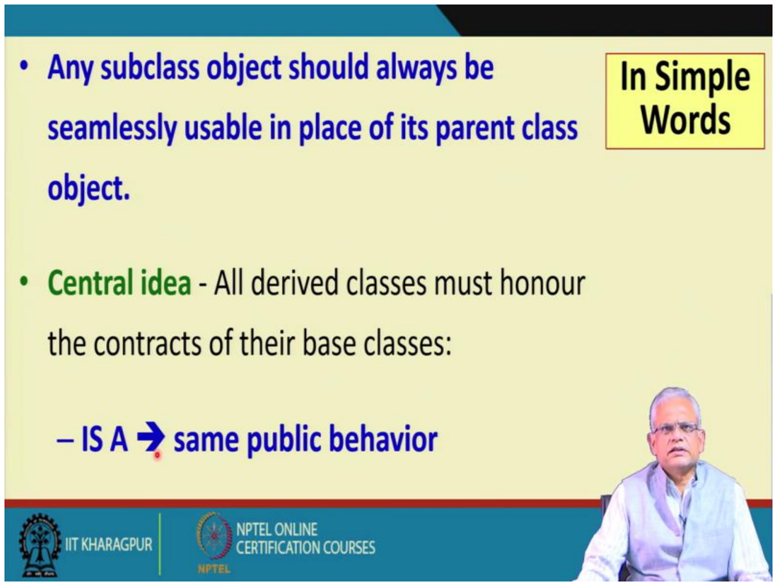
Liskov Substitution Principle

- Statement: "An instance of a derived class should be able to replace any instance of its super class"
- In effect, the base class establishes a basic behavior (functionality) that clients know of.
  - If a derived class violates the contract, then a violation of LSP occurs.

Student
UG
PG
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

An example here is that a student is the base class and the UG and PG are the derived classes. Now, if I have an array of students, then there may be UG, PG objects in the student array because it is possible to have instances of UG, PG as elements of the student array. But a client of the student array might use different functionalities, for example, print, if the print of these two are different UG, PG then we cannot do it seamlessly or to first identify which object is residing at an element in the array and based on that you will have to do it and that would be a violation of the Liskov Substitution Principle.

(Refer Slide Time: 15:10)



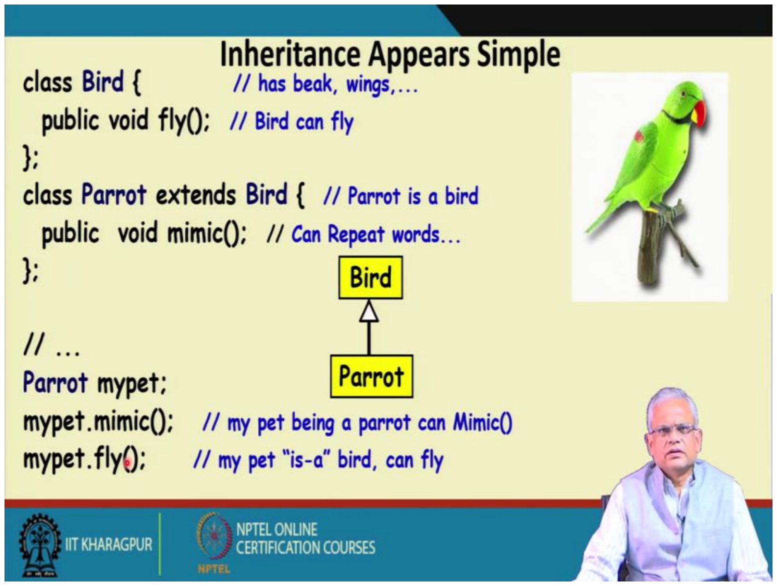
In Simple Words

- Any subclass object should always be seamlessly usable in place of its parent class object.
- Central idea - All derived classes must honour the contracts of their base classes:
  - IS A → same public behavior

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The Liskov substitution principle in simple words is that a subclass object should always be seamlessly usable in place of its parent class. This is possible in a typical situation because the derived classes honour the basic behavior of the base classes and that is why we have been saying that the inheritance relation is also known as a IS A relation. A UG student is a student, a PG student is also a student so that is the IS A relation which implies the same public behavior.

(Refer Slide Time: 16:00)



Inheritance Appears Simple

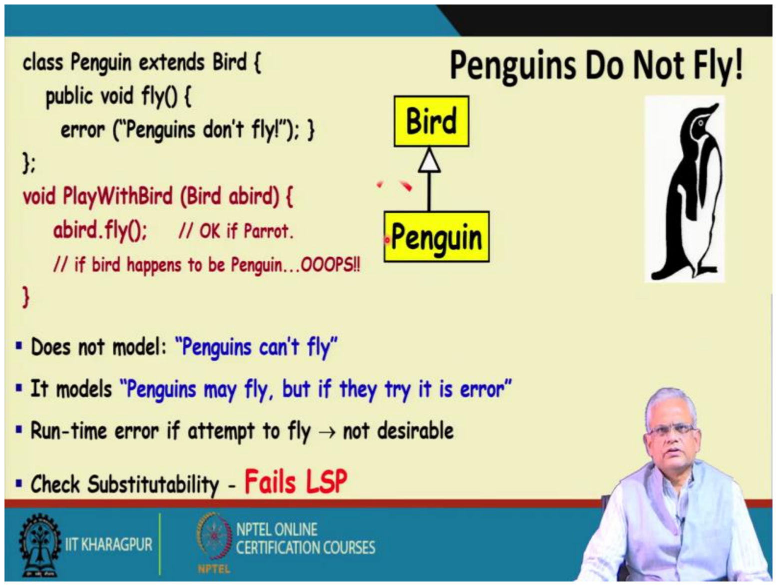
```
class Bird {            // has beak, wings,...
  public void fly();    // Bird can fly
};
class Parrot extends Bird {   // Parrot is a bird
  public  void mimic();       // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();   // my pet being a parrot can Mimic()
mypet.fly();     // my pet "is-a" bird, can fly
```

Bird
Parrot
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, let us see the situations where misuse can occur and there will be a violation of the Liskov substitution principle and if there is a violation of the Liskov substitution principle, there will be unnecessary complexity of the code, it will be difficult to understand and maintain. Let us look at the simple code.

We have the class bird and here we have this public method fly because birds can fly. Now, we can have a derived class parrot here which extends the base class bird and then it adds additional methods like mimic, it inherits the fly method and also adds additional methods like mimic. And then we can create an instance of parrot and then we can call the mimic here, fly here and it works fine.

(Refer Slide Time: 17:14)



Penguins Do Not Fly!

```
class Penguin extends Bird {
   public void fly() {
     error ("Penguins don't fly!"); }
};
void PlayWithBird (Bird abird) {
    abird.fly();     // OK if Parrot.
    // if bird happens to be Penguin...OOOPS!!
}
```

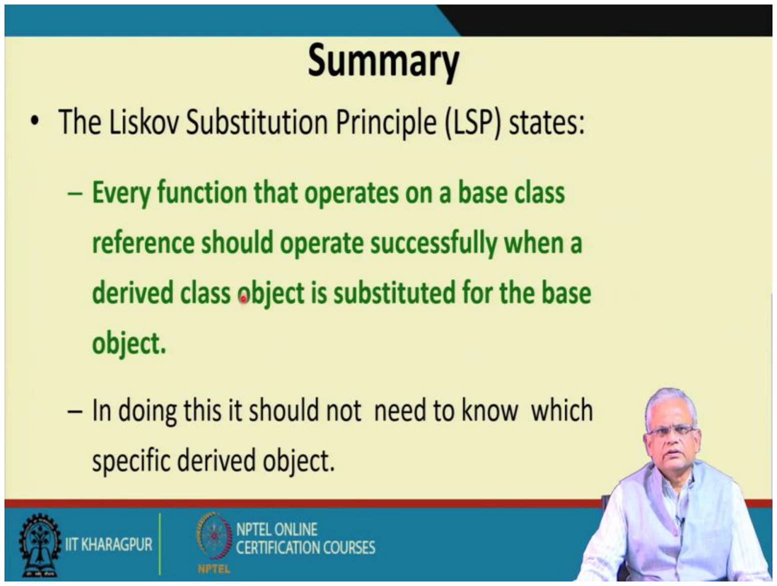
- Does not model: "Penguins can't fly"
- It models "Penguins may fly, but if they try it is error"
- Run-time error if attempt to fly → not desirable
- Check Substitutability - Fails LSP

Bird
Penguin
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

But suppose we use penguin as a derived class of bird but we know that the penguins do not fly actually. And then once we inherit the bird in the penguin class, penguin extends bird, then for the fly we override the fly and write here that “penguins do not fly”. And the fly method works for all birds and does not work if the bird happens to be a penguin.

And this is an example of a violation of the Liskov substitution principle, the derived class is not complaint to the behavior of the base class. We cannot substitute a penguin for where a bird is required, an object of bird is required we cannot have a penguin object there. And therefore, this is an example of a failure of the Liskov substitution principle.

(Refer Slide Time: 18:43)



Summary

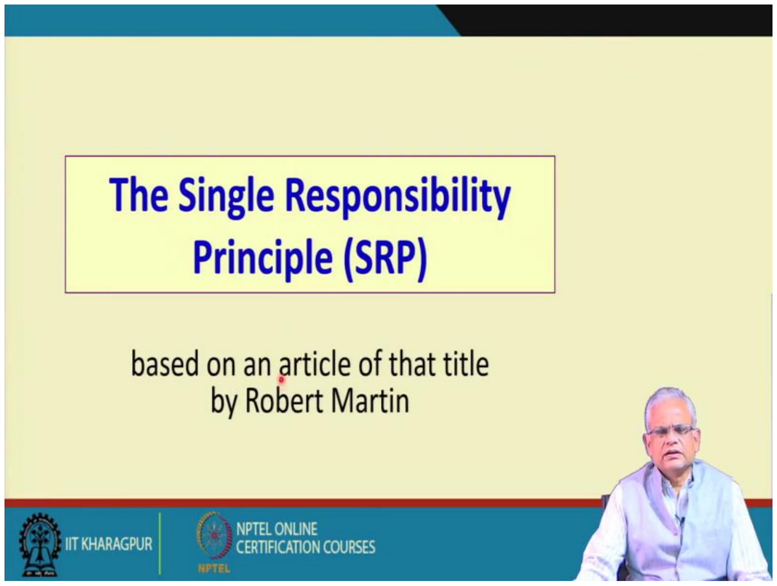
- The Liskov Substitution Principle (LSP) states:
  - Every function that operates on a base class reference should operate successfully when a derived class object is substituted for the base object.
  - In doing this it should not need to know which specific derived object.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The Liskov substitution principle states that every function that operates on a base class reference should operate successfully on a derived class object. Typically happens in a good inheritance hierarchy. But if we are not careful, it may so happen that the derived class is not complaint to the base class and therefore leads to extra complexity.

The client needs to know which object is dealing with rather than dealing all objects transparently, we will have to identify the object type and then deal with it lead to extra complexity.

(Refer Slide Time: 19:32)



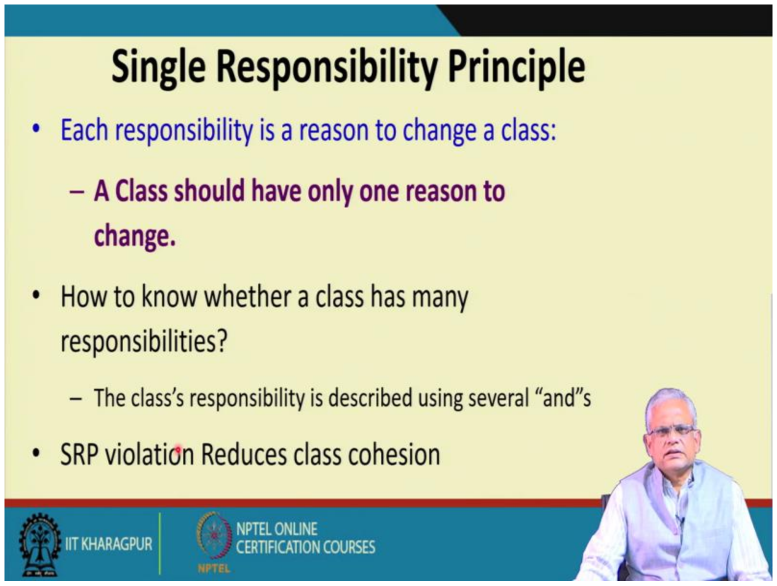
The Single Responsibility Principle (SRP)

based on an article of that title by Robert Martin

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Now, let us look at the third principle which is the Single Responsibility Principle or the SRP. This is again based on article by Robert Martin.

(Refer Slide Time: 19:50)



Single Responsibility Principle

- Each responsibility is a reason to change a class:
  - A Class should have only one reason to change.
- How to know whether a class has many responsibilities?
  - The class's responsibility is described using several "and"s
- SRP violation Reduces class cohesion

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

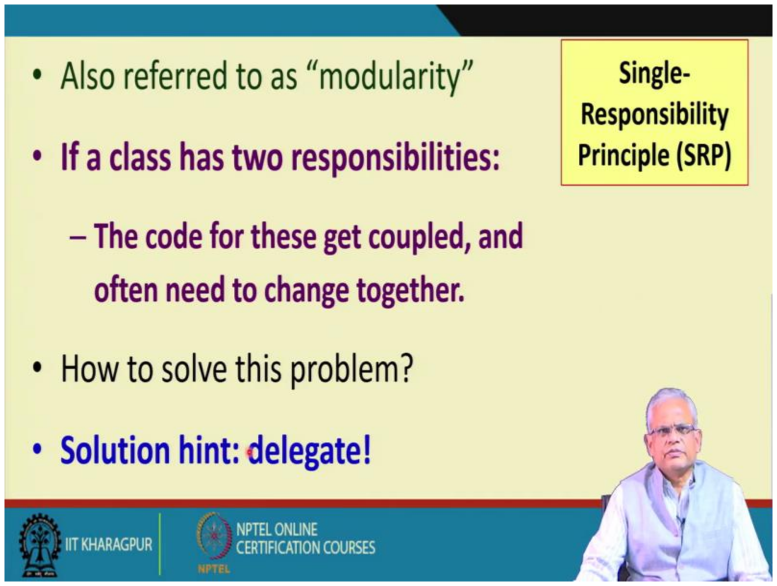
The principle applies to the responsibility of a class and we know that responsibility is basically the method supported by the class and if a class needs to change, it is because of the responsibilities or the methods it supports, the methods may need to change, for performance reasons or for making it applicable to certain situation and so on.

Given that the class needs to change because of the responsibilities if it has multiple responsibilities it will need more changes. But how do we know that a class has a single responsibility or multiple responsibility? A responsibility is can be implemented by a set of methods and the set of methods achieve something coherent.

For example, we may say that this set of methods handle the student admission and this set of methods handle the accounts. If we want to state the responsibility of a class and we use several "and"s that means it is having more than one responsibility, just like we said that there is a student admission, the student's grading and the student's account and so on.

If we have too many responsibilities with a class, then the class will need frequent changes and will likely to have bugs. And also these changes will force the client classes of this class to change even though they might be using some responsibility of the class which has not changed. To think of it, the single responsibility principle states that classes should be cohesive, they should have a single responsibility and if there is a violation of SRP then the classes have low cohesion.

(Refer Slide Time: 22:38)



Single-Responsibility Principle (SRP)

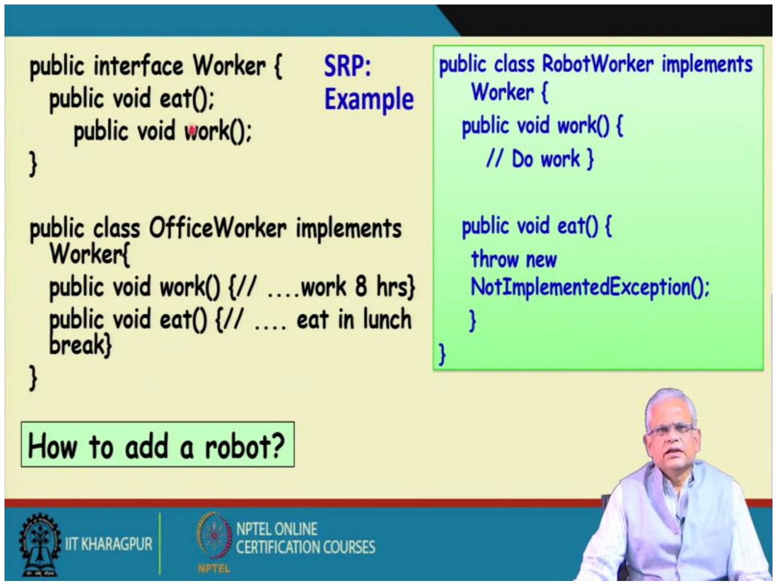
- Also referred to as "modularity"
- **If a class has two responsibilities:**
  - **The code for these get coupled, and often need to change together.**
- How to solve this problem?
- **Solution hint: delegate!**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The single responsibility principle also is known as "modularity" principle. If a code has more than one responsibility, then each responsibility will need change for different reasons and the problem here is that the code gets so coupled that one responsibility may require changes to the other responsibility of the class.

But the question now is that how do we solve this problem if we have designed a class which has more than one responsibility? The solution here is to delegate, it should do one responsibility and the other responsibilities it should have a separate class of which it can take help or delegate those responsibilities if it has multiple responsibilities it should achieve only one of the central responsibility the other responsibilities it should delegate.

(Refer Slide Time: 23:51)



SRP: Example

```
public interface Worker {
   public void eat();
      public void work();
}

public class OfficeWorker implements
   Worker{
   public void work() {// ....work 8 hrs}
   public void eat() {// .... eat in lunch
   break}
}
```

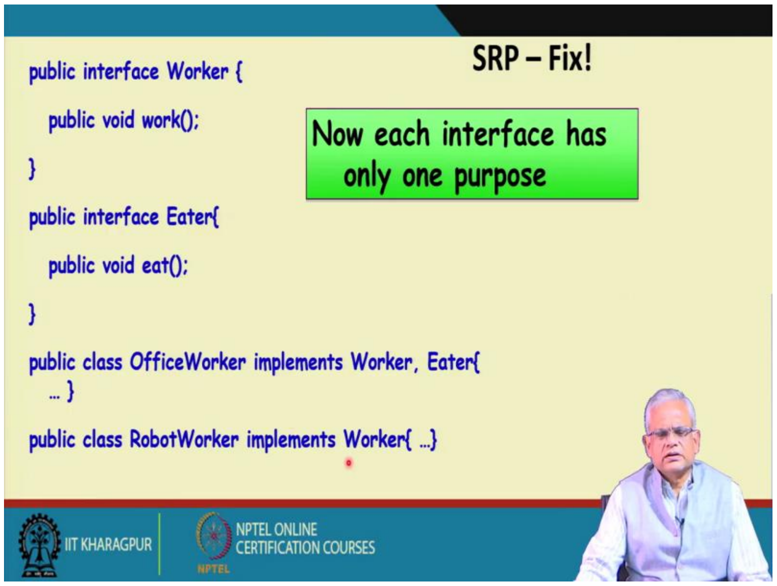
```
public class RobotWorker implements
   Worker {
  public void work() {
     // Do work }

  public void eat() {
   throw new
   NotImplementedException();
   }
}
```

How to add a robot?

Let us just take an example. Here we have this interface worker and there are two method prototypes here, eat and work. Now, we have this concrete class office worker implements the worker interface and we have this work we provided the body of the method work for 8 hours and eat it the lunch break.

But now let us say we have a special type of worker which is a robot. Now, the robot should implement the worker interface and when we try to implement the worker interface we need to provide definitions for work, we write that. But what about eat? Robots do not eat and therefore will just write throw new not implemented exception. The problem here, if we think of it, is that the problem lies with the worker interface which has two different responsibilities eat and work.

(Refer Slide Time: 25:48)



SRP – Fix!

```
public interface Worker {
   public void work();
}
public interface Eater{
   public void eat();
}
public class OfficeWorker implements Worker, Eater{
   … }
public class RobotWorker implements Worker{ …}
```

Now each interface has only one purpose

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

And to make it complaint with the single responsibility principle we need to separate out these two responsibilities, we have the interface worker and the interface eater. And the office worker will implement both the worker and eater interface and the Robot worker implements the worker. And now each interface has only one purpose and it is complaint with the SRP principle.

Without the code being SRP complaint we will have ugly code which is difficult to understand, difficult to maintain, there will be too many bugs and we just saw the solution to the SRP problem is to separate out the different interfaces and also to delegate if we need some responsibility or multiple responsibilities from an object we need to delegate some of the responsibility to another class. We are almost at the end of this session, we will stop here and continue in the next session. Thank you.