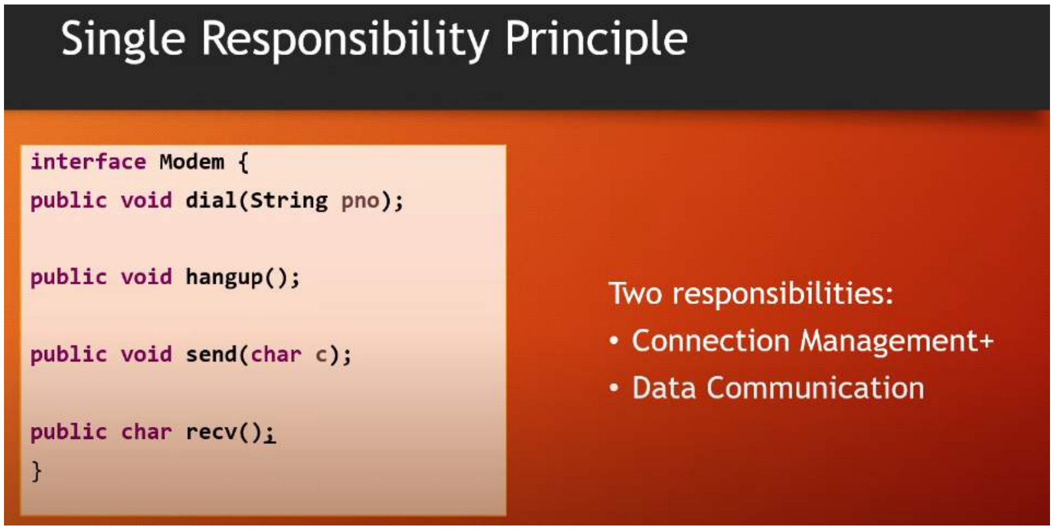
S = SINGLE RESPONSIBILITY
O = OPEN CLOSED
L = LISKOV SUBSTITUTION
I = INTERFACE SEGREGATION
D = DEPENDENCY INVERSION

<u>SINGLE RESPONSIBILITY</u>

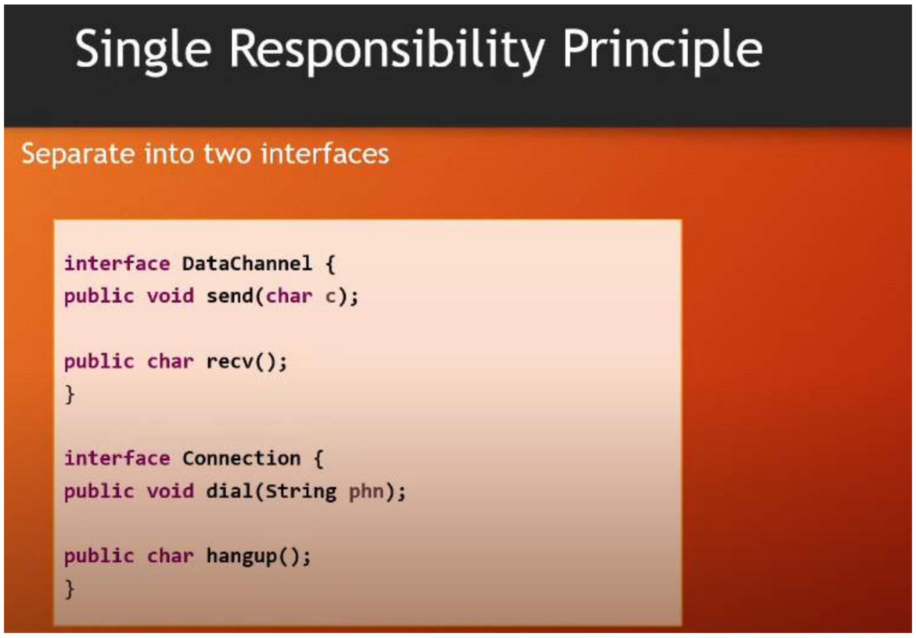


Single Responsibility Principle

```
interface Modem {
public void dial(String pno);

public void hangup();

public void send(char c);

public char recv();
}
```

Two responsibilities:
- Connection Management+
- Data Communication

Above interface serves two responsibilities. One is connection management (dial and hang up)and the other is Data communication(send and recv).



Single Responsibility Principle

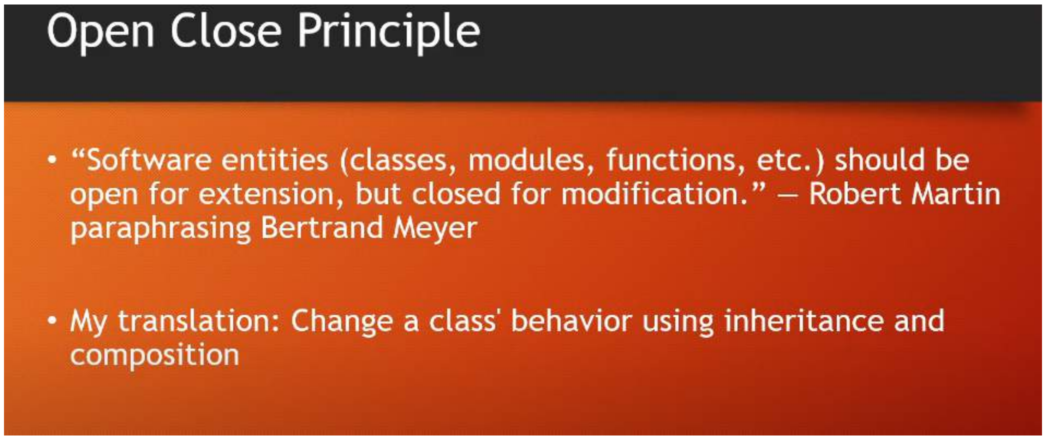
Separate into two interfaces

```
interface DataChannel {
public void send(char c);

public char recv();
}

interface Connection {
public void dial(String phn);

public char hangup();
}
```

This one is a correction in the Modem interface. Now we have two separate interfaces for two separate responsibilities.

Project Scenario :-
We have separate controllers, service and repo layer for separate features.

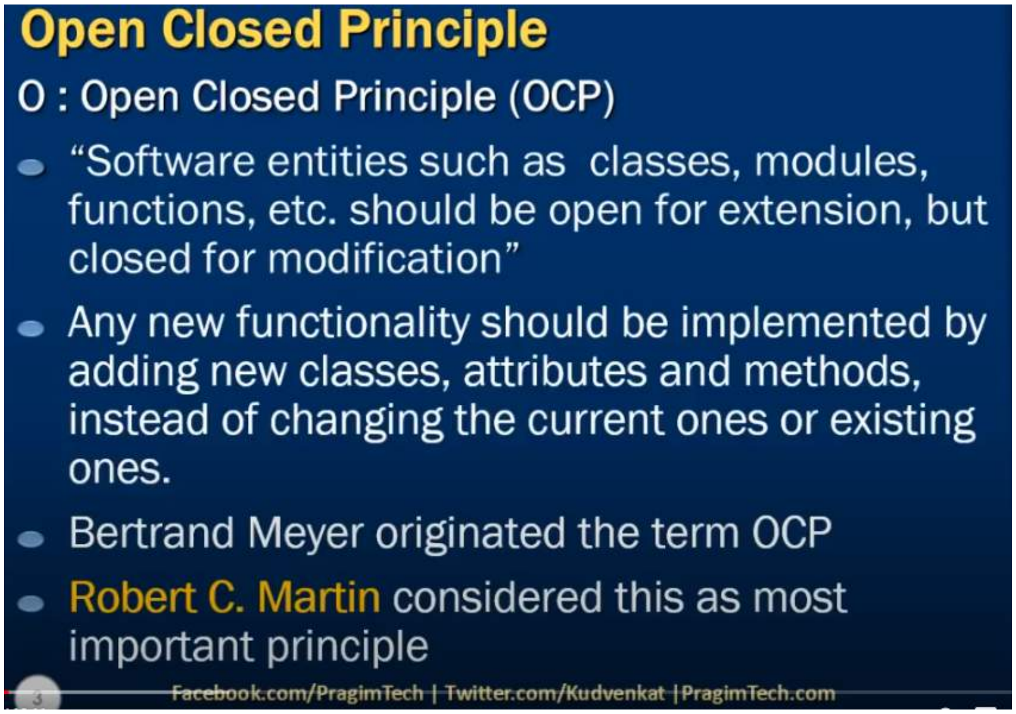
<u>OPEN CLOSED</u>



Open Close Principle

- "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." — Robert Martin paraphrasing Bertrand Meyer
- My translation: Change a class' behavior using inheritance and composition

Definition 2:

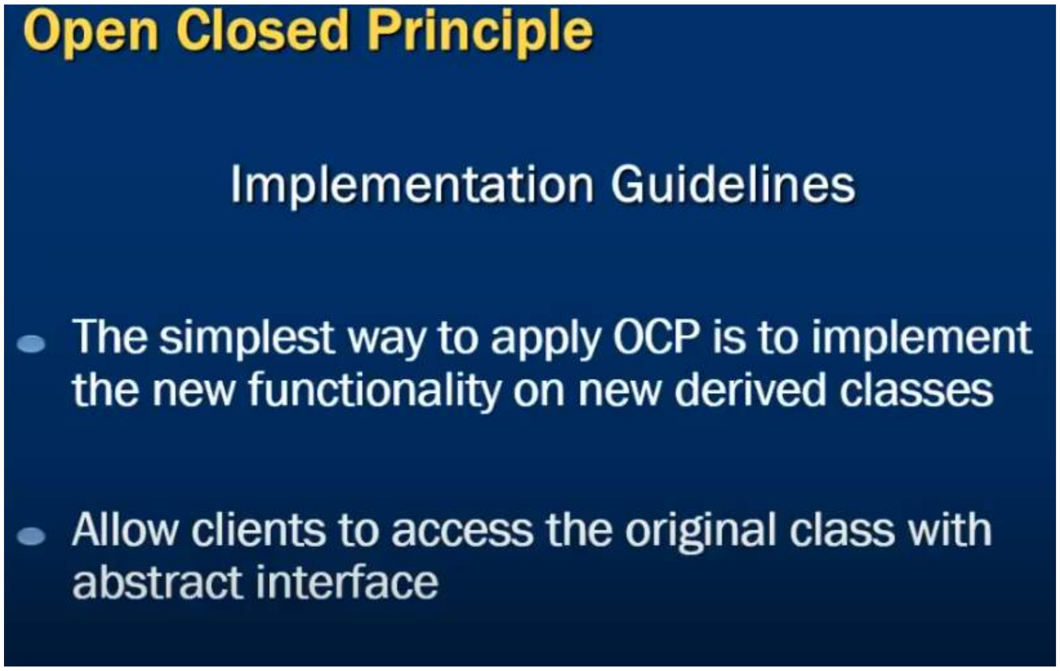


Open Closed Principle

O : Open Closed Principle (OCP)

- "Software entities such as classes, modules, functions, etc. should be open for extension, but closed for modification"
- Any new functionality should be implemented by adding new classes, attributes and methods, instead of changing the current ones or existing ones.
- Bertrand Meyer originated the term OCP
- Robert C. Martin considered this as most important principle

Facebook.com/PragimTech | Twitter.com/Kudvenkat | PragimTech.com



Open Closed Principle

Implementation Guidelines

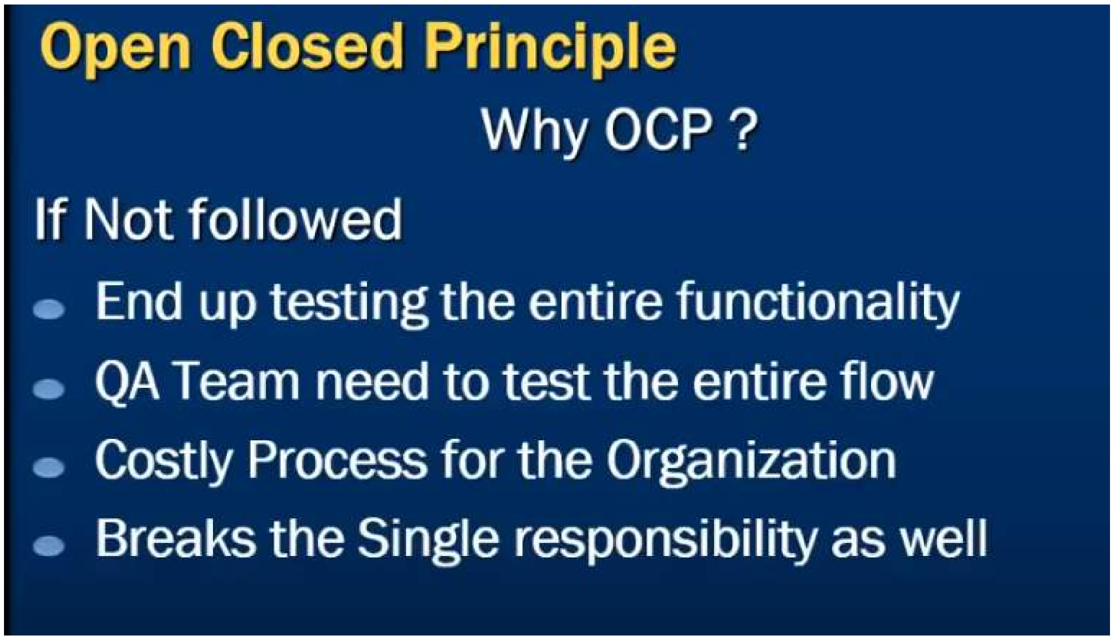
- The simplest way to apply OCP is to implement the new functionality on new derived classes
- Allow clients to access the original class with abstract interface

If we want to add any new functionality or feature to the existing application, we must leave the original implementation untouched.
If we have to change the class in order to add the new functionality, better extend that class to a new class.
If we have to change a particular method, better create a new method and call the old method for its output and use that output to generate new functionality.

Use Case :-
For example, suppose we have some currencyConverterMethod, which is being called for IndianCurrency and UsdCurrency. So here we are not changing currencyConverterMethod, rather we are extending the functionality of currencyConverterMethod for calculating IndianCurrency and UsdCurrency.

```
Class CurrencyConvert {

Long currencyConverterMethod() {
}

Long IndianCurrency() {
Long I = currencyConverterMethod();
….
}
Long UsdCurrency() {
Long I = currencyConverterMethod();
….
}
```

}



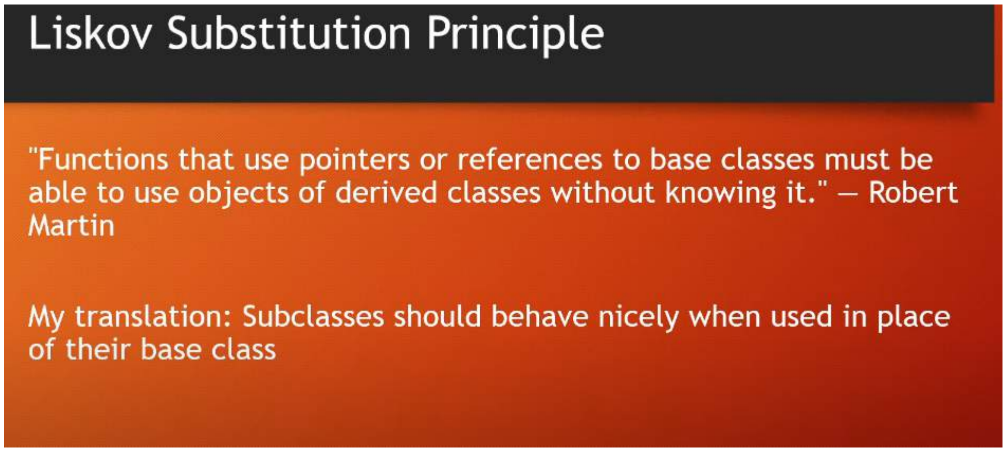
Open Closed Principle

Why OCP ?

If Not followed

- End up testing the entire functionality
- QA Team need to test the entire flow
- Costly Process for the Organization
- Breaks the Single responsibility as well

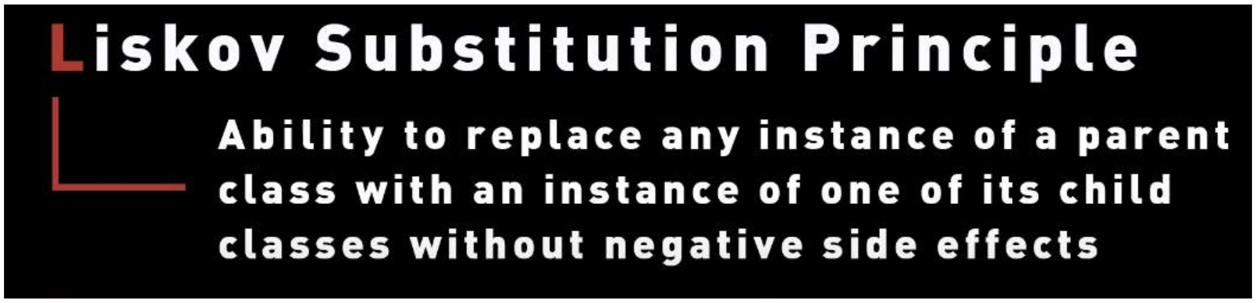
<u>LISKOV SUBSTITUTION PRINCIPLE</u>



Liskov Substitution Principle

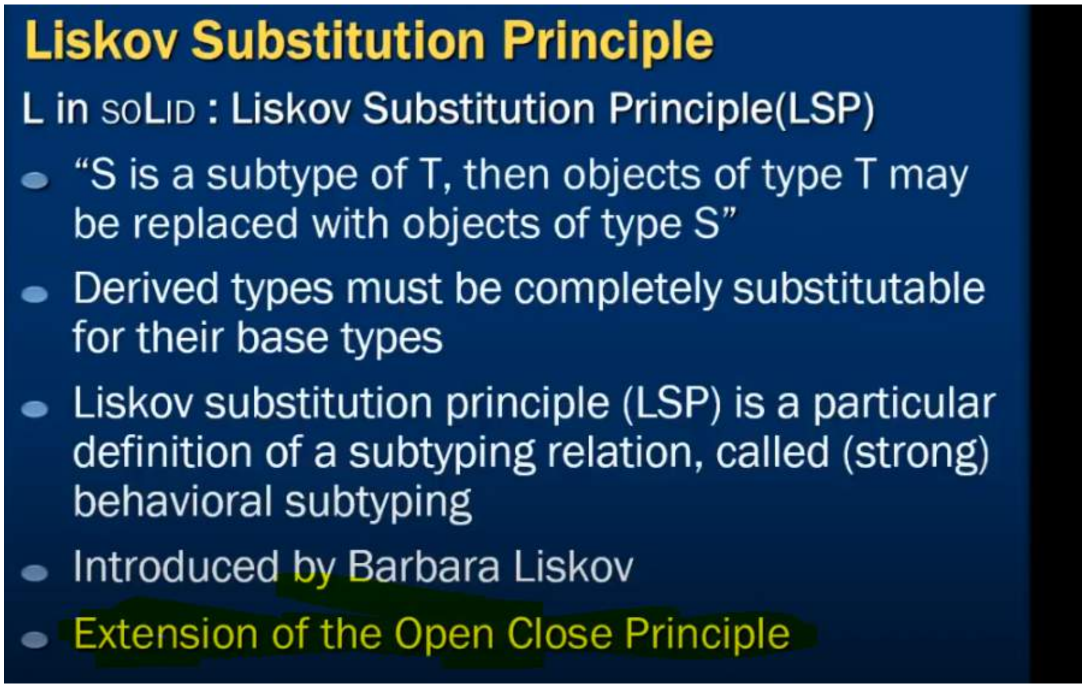
"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." — Robert Martin

My translation: Subclasses should behave nicely when used in place of their base class



Liskov Substitution Principle

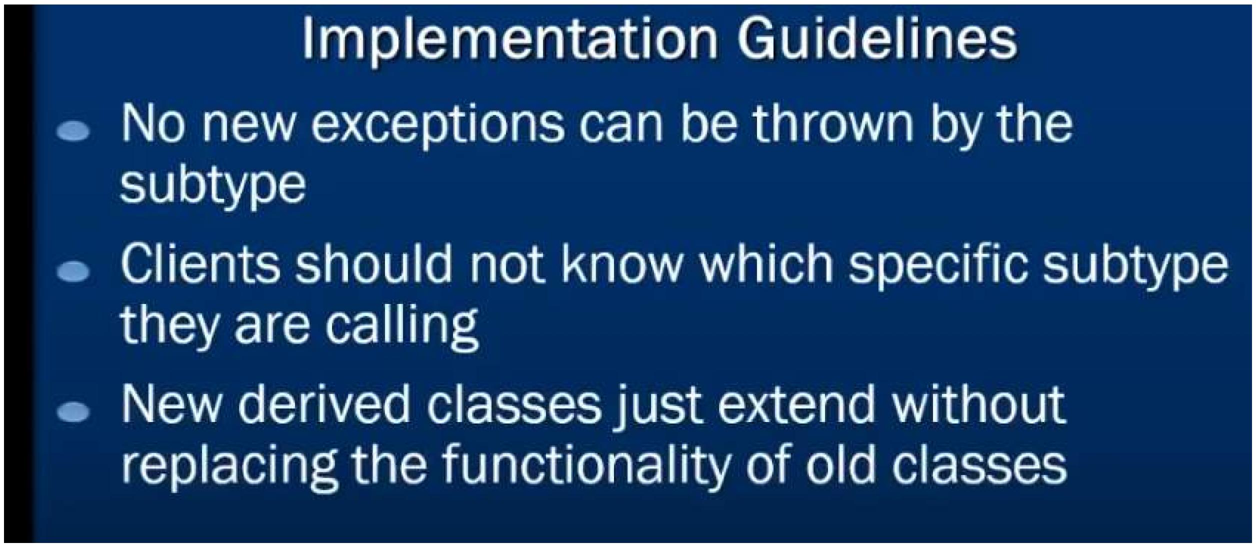
Ability to replace any instance of a parent class with an instance of one of its child classes without negative side effects



Liskov Substitution Principle

L in soLID : Liskov Substitution Principle(LSP)

- "S is a subtype of T, then objects of type T may be replaced with objects of type S"
- Derived types must be completely substitutable for their base types
- Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping
- Introduced by Barbara Liskov
- Extension of the Open Close Principle



Implementation Guidelines

- No new exceptions can be thrown by the subtype
- Clients should not know which specific subtype they are calling
- New derived classes just extend without replacing the functionality of old classes

Or in other languages, if class A is a subtype of class B, we should be able to replace B with A without disrupting the behaviour of our program.

```
Or A -|
        B-|
            C-|
```
If we have such inheritance relationships between classes where A is the parent class and B and C are the child classes respectively. According to the principle if our class is extending A then we should easily be able to replace A with any of it’s child classes. In this case B or C. Without any hassle.

Or in other words, it’s saying that everything should be kind of a plug and play.

Or It must follow the Adapter pattern.

Or a kind of ORM tool. With the help of ORM tools such as Hibernate. We can easily replace any database without much hassle in our code. It is kind of a plug and play.

--------------------------Most suitable use case-----------------------------------

```
interface Shape {
        Int calculateArea();
}

class Rectangle implements Shape {
        Int calculateArea() {

        }
}



class Square implements Shape {
        Int calculateArea() {

        }
}

public class LiskovSPDemo {
        public static void main(String[] args) {
                Shape  rectangle = new Rectangle();
                Shape  square = new Square ();

                rectangle.calculateArea();
                square.calculateArea();

        }
}
```

In the above implementation we have Shape as the super type of both Rectangle and Square. And in our client, we are able to easily replace Square with Square without much hassle. Also there is no need to change the method name here, although both shapes have separate implementations of it.

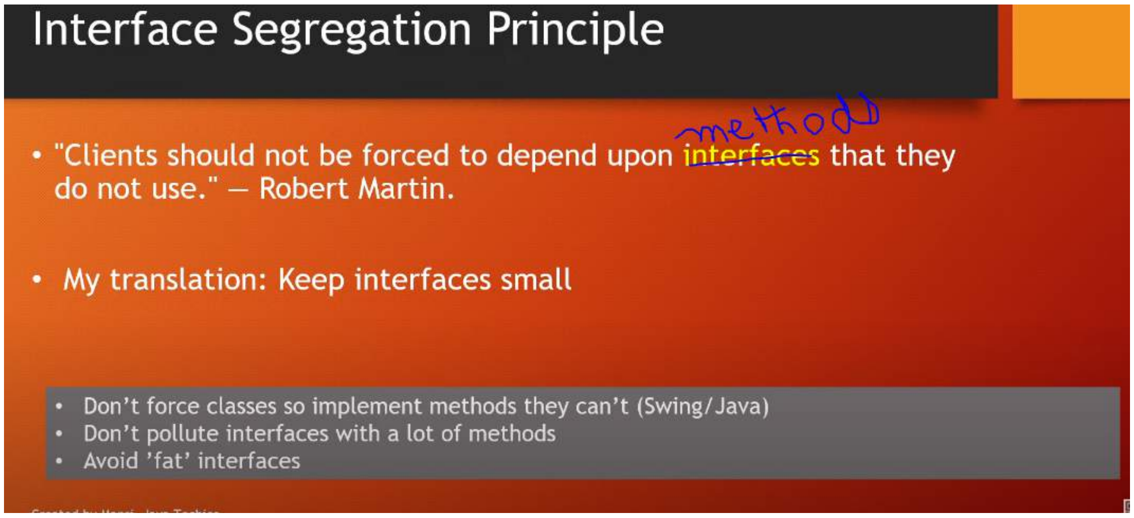
Point to be noted :- Here we must be aware of the exception chaining principle Java, especially rule 2. Same principle also applies to the LSP.
There two main rules to note here :-

1) **If the superclass method does not declare an exception**
    - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
2) **If the superclass method declares an exception**
    - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

We must be aware of rule 2 in lsp.
--------------------------Most suitable use case ends------------------------------------

<u>INTERFACE SEGREGATION PRINCIPLE</u>
Formal definition (correction from the below diagram) - **no client should be forced to depend on methods it does not use**.

It basically talks about keeping the interfaces clean and segregated (that means loose coupled).



Interface Segregation Principle

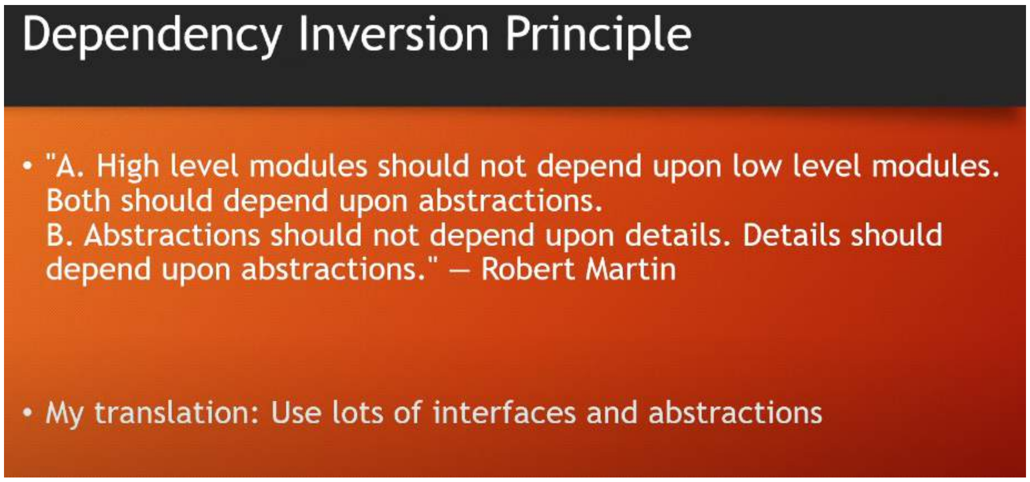
- "Clients should not be forced to depend upon ~~interfaces~~ methods that they do not use." – Robert Martin.
- My translation: Keep interfaces small
    - Don't force classes so implement methods they can't (Swing/Java)
    - Don't pollute interfaces with a lot of methods
    - Avoid 'fat' interfaces

Bad Implementation :-

Or another example is,

```
interface Shape {
	Int calculateArea();
	Int calculateLabourCost();//Not need
}

class Rectangle implements Shape {
	Int calculateArea() {

	}

	Int calculateLabourCost() {} //Unnecessary method. It is overridden with empty
functionality just to stop compilation errors. Although not needed.

}



class Square implements Shape {
	Int calculateArea() {

	}
Int calculateLabourCost() {} //Unnecessary method. It is overridden with empty functionality just
to stop compilation errors. Although not needed.
}

Correction :-
interface Shape {
        Int calculateArea();
}

class Rectangle implements Shape {
        Int calculateArea() {

        }
}



class Square implements Shape {
        Int calculateArea() {

        }
}
```

<u>DEPENDENCY INVERSION PRINCIPLE</u>



Dependency Inversion Principle

- "A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
  B. Abstractions should not depend upon details. Details should depend upon abstractions." — Robert Martin

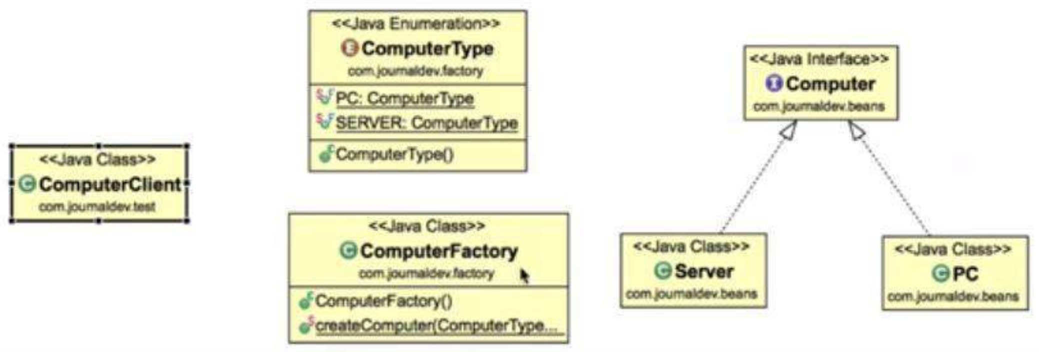
- My translation: Use lots of interfaces and abstractions

Basically it is saying to use factory as well as AbstractFactory design patterns.

Common example is Spring IOC. Where bean control is in the spring container.

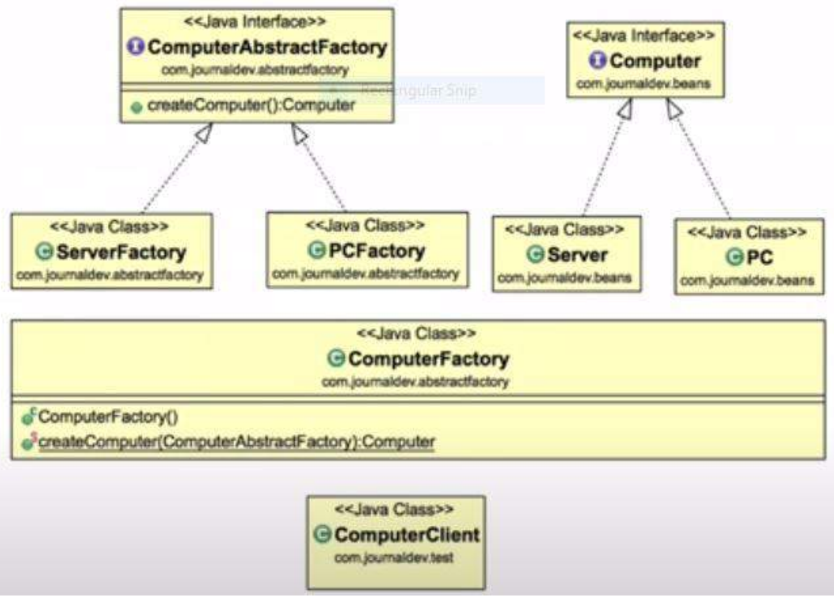
You can take examples of JournalDev factory and abstractFactory use cases.

How to implement Factory Pattern?



```
Computer pc = ComputerFactory.createComputer(ComputerType.PC, "1GB", "256GB", "Intel Core 2 Duo");
Computer server = ComputerFactory.createComputer(ComputerType.SERVER, "32GB", "1TB", "AMD Ryzen 7 2700X");
```

Abstract Factory Implementation


<<Java Interface>>
ComputerAbstractFactory
com.journaldev.abstractfactory
createComputer():Computer
<<Java Interface>>
Computer
com.journaldev.beans
<<Java Class>>
ServerFactory
com.journaldev.abstractfactory
<<Java Class>>
PCFactory
com.journaldev.abstractfactory
<<Java Class>>
Server
com.journaldev.beans
<<Java Class>>
PC
com.journaldev.beans
<<Java Class>>
ComputerFactory
com.journaldev.abstractfactory
ComputerFactory()
createComputer(ComputerAbstractFactory):Computer
<<Java Class>>
ComputerClient
com.journaldev.test