PEMROGRAMAN LANJUT

SOLID: ISP dan DIP

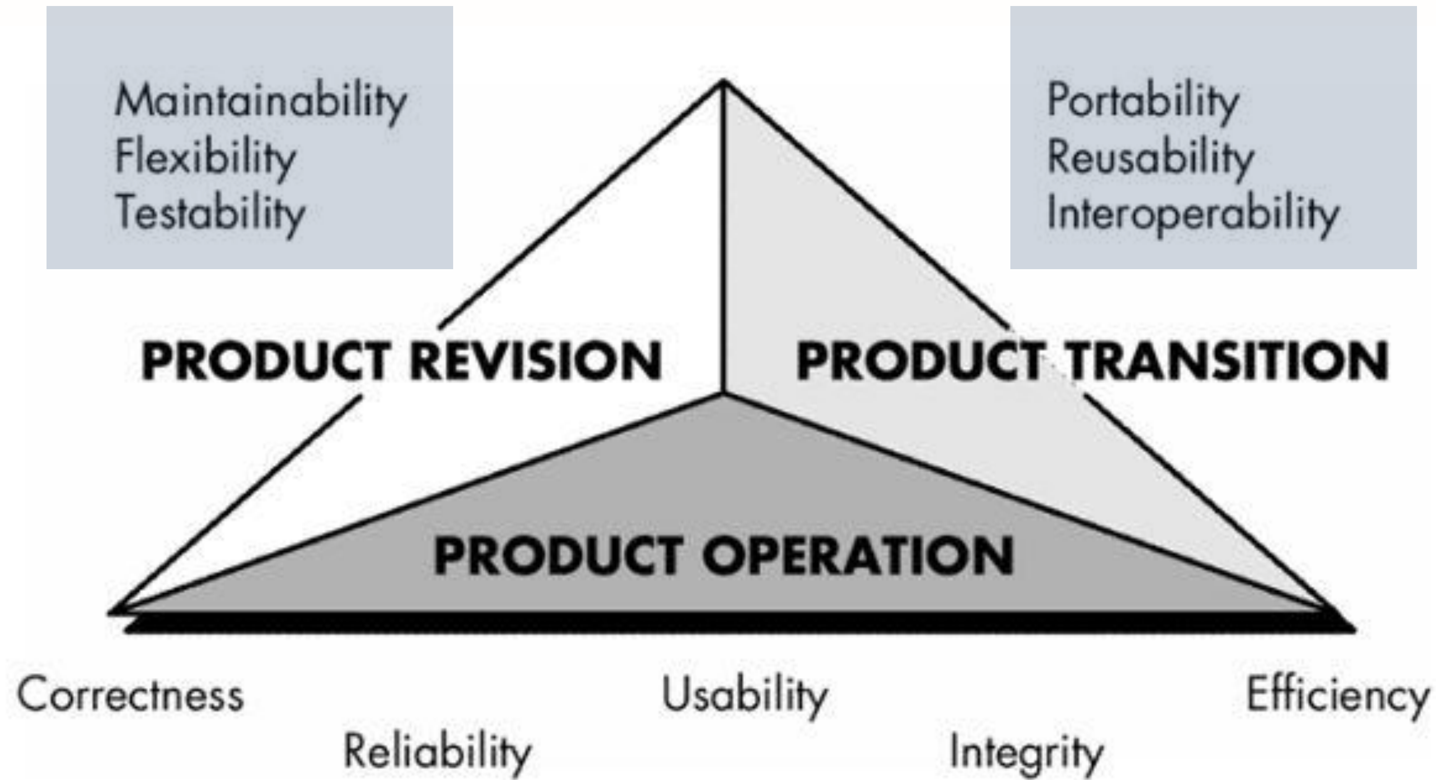
Oleh

Tri Hadiah Muliawati

Politeknik Elektronika Negeri Surabaya

2021

Review



Mc Call Software Quality Metric

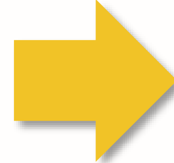
Open Close Principle

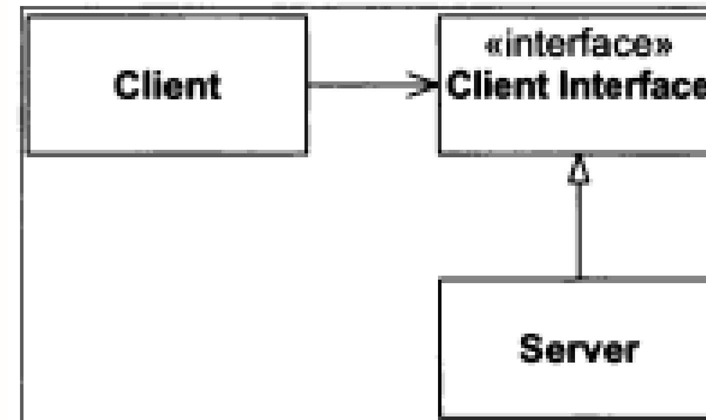
- Originated by Bertrand Meyer, 1988
- Software entities should be open for extension, but closed for modification
- A good software architecture would reduce the amount of changed code to the barest minimum. Ideally, zero.

Open Close Principle

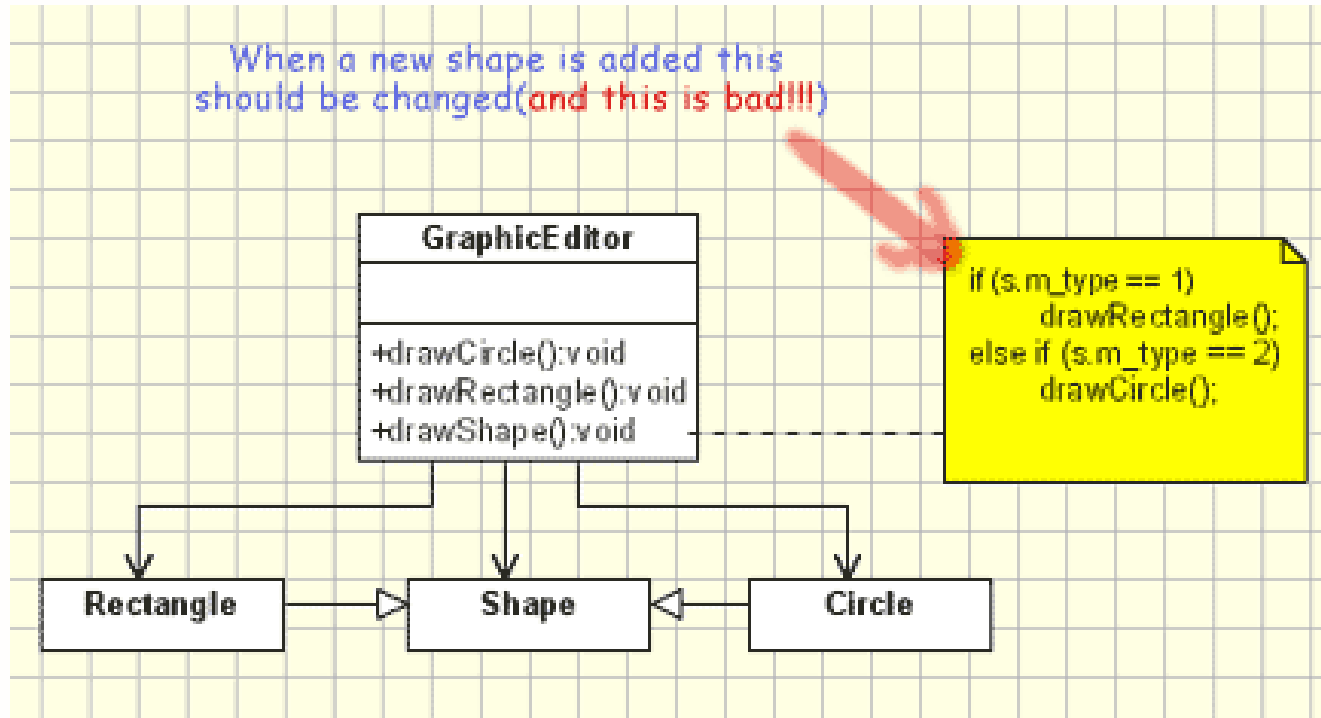
- It is implemented using abstraction

By using abstraction, **Client class is not tightly coupled with Server class**. Thus, if there is a new requirement to add another type of Server, developer can create new class as long as it implements Client Interface.





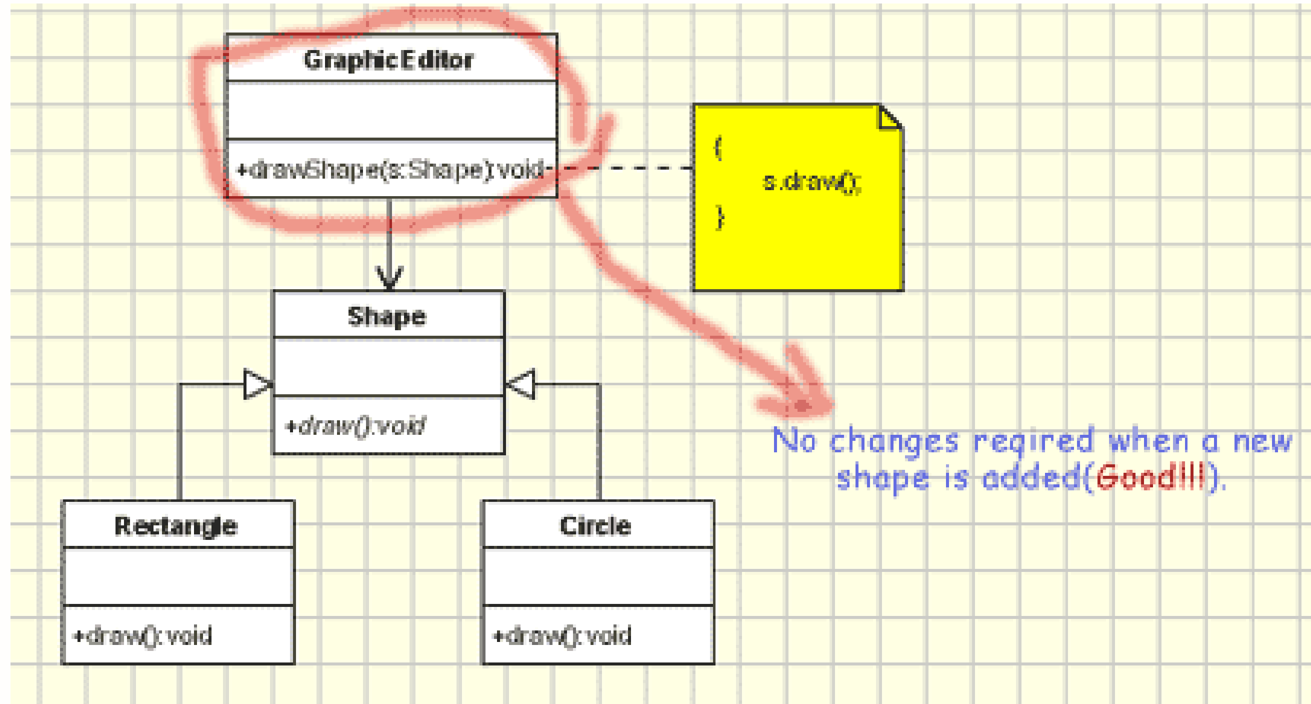
Open Close Principle



If there is new requirement to add another type of shape (i.e.: Triangle, Trapezium, Pentagon, Diamond, etc.), developer should modify drawShape() in GraphicEditor. **It is a violation of OCP**.

```
// Open-Close Principle - Bad example
 class GraphicEditor {

        public void drawShape(Shape s) {
                if (s.m_type==1)
                        drawRectangle(s);
                else if (s.m_type==2)
                        drawCircle(s);
        }
        public void drawCircle(Circle r) {....}
        public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
        int m_type;
 }

 class Rectangle extends Shape {
        Rectangle() {
                super.m_type=1;
        }
 }

 class Circle extends Shape {
        Circle() {
                super.m_type=2;
        }
 }
```

Open Close Principle




```
// Open-Close Principle - Good example
 class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
 }

 class Shape {
        abstract void draw();
 }

 class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
 }
```

If there is new requirement to add another type of shape (i.e.: Triangle, Trapezium, Pentagon, Diamond, etc.), developer can easily create a new class which is an extension of Shape class. **It is an implementation of OCP**.

Open Close Principle

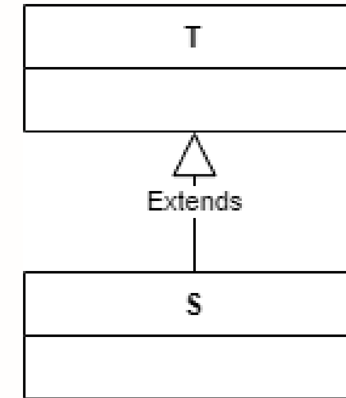
- Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed.
- We can adapt “Fool Me Once” to keep loading our software with needless complexity. Thus, we initially write our code expecting it not to change. When change occurs, we implement the abstractions that protect us from future changes of that kind.

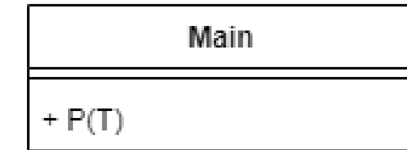
Liskov Substitution Principle

- Originated by Barbara Liskov, 1988
- Subtypes must be substitutable for their base types.
- We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Liskov Substitution Principle

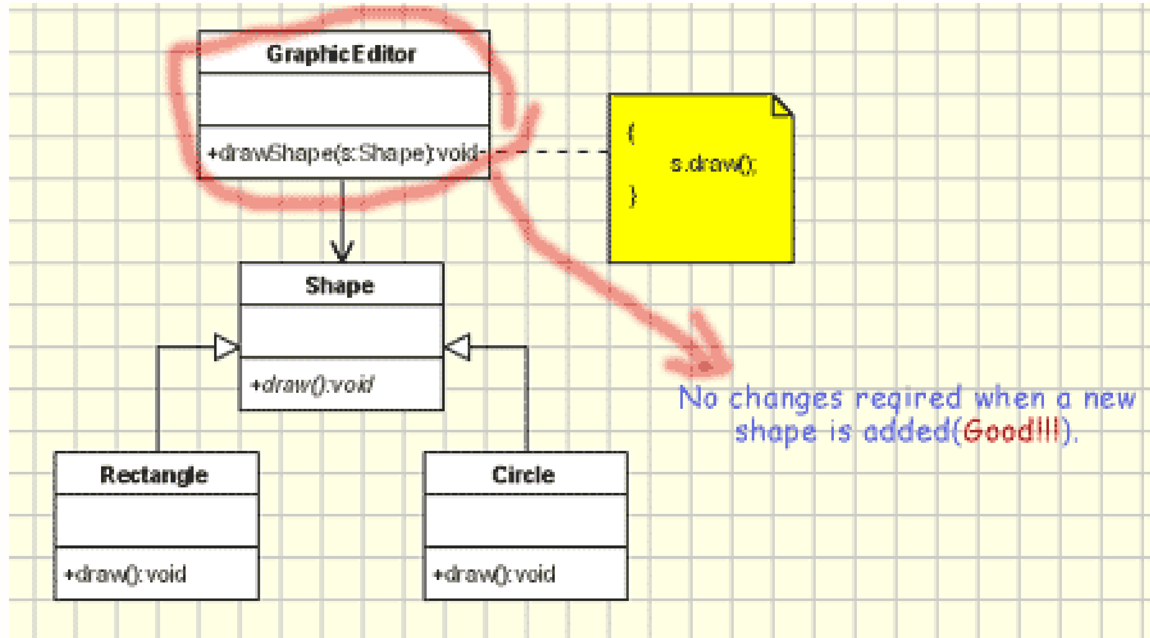
- If for each object `O1` of type `S`, there is an object `O2` of type `T`. Such that for all programs `P` defined in terms of the `T`, the behavior of `P` is unchanged when `O1` is substituted for `O2` when `S` is a subtype of `T`.



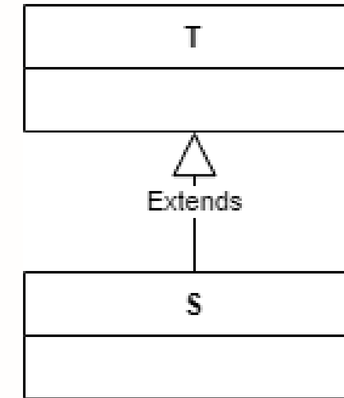



```
Main{
    main(){
        S O1 = new S();
        T O2 = new T();
        P(O1);
        P(O2);
    }
}
```

Liskov Substitution Principle



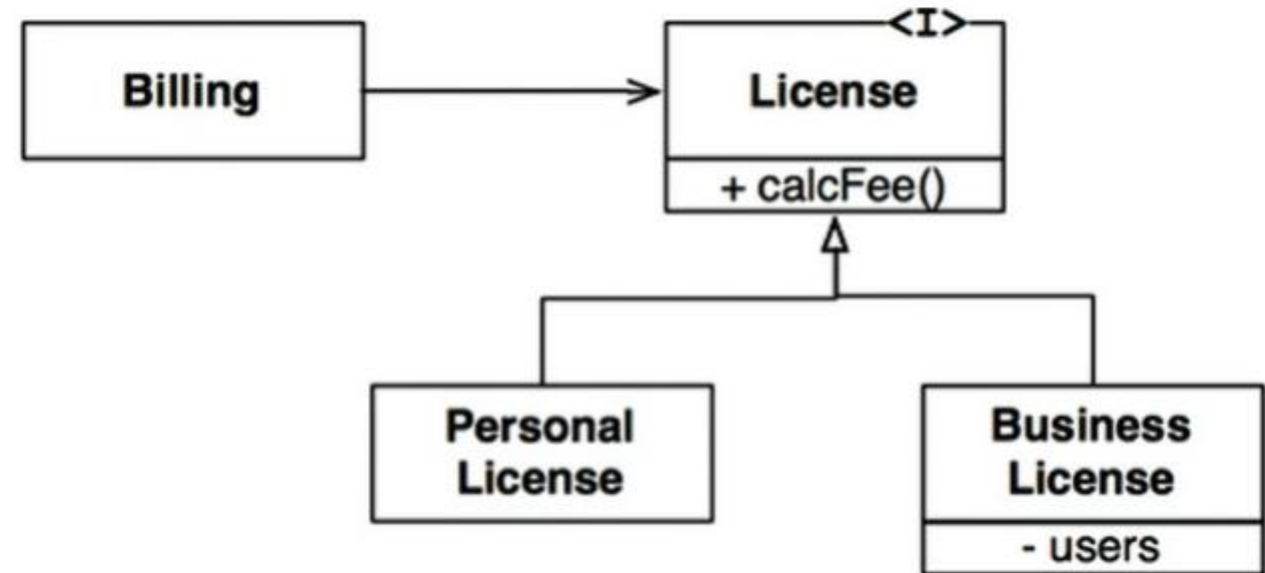
The behavior of drawShape() is unchanged when object of class Shape is substituted by object of class Rectangle or class Circle



```
Main{
    main(){
        S O1 = new S();
        T O2 = new T();
        P(O1);
        P(O2);
    }
}
```

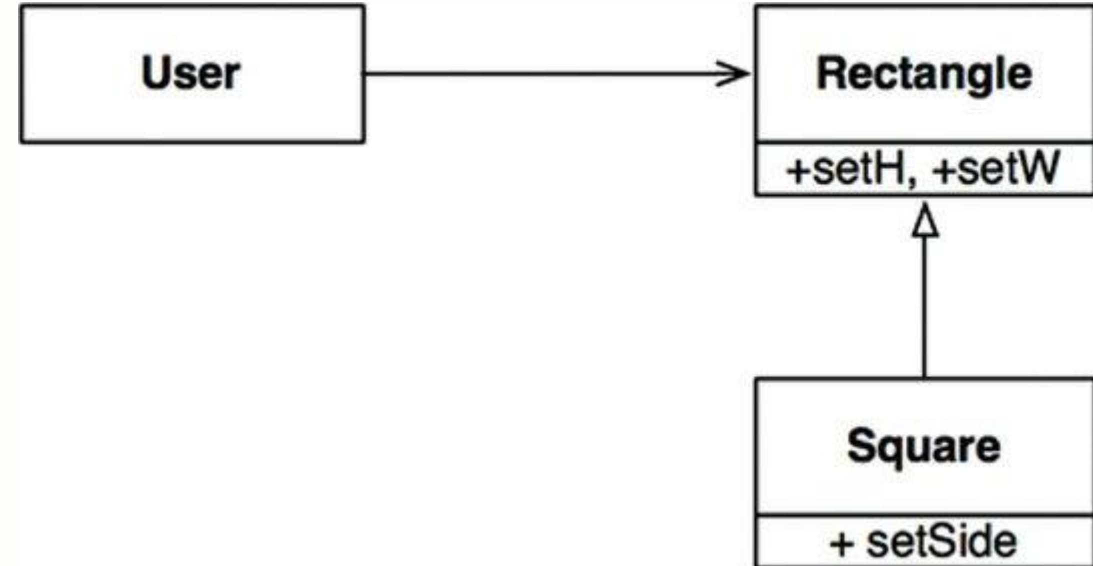
Liskov Substitution Principle

- Billing class has a method named calcFee(), which is called by the Billing application.
- There are two “subtypes” of License: PersonalLicense and BusinessLicense. They use different algorithms to calculate the license fee.
- This design conforms to the LSP because the behavior of the Billing application does not depend, in any way, on which of the two subtypes it uses. Both of the subtypes are substitutable for the License type



Liskov Substitution Principle

- In this example, Square is not a proper subtype of Rectangle because the height and width of the Rectangle are independently mutable; in contrast, the height and width of the Square must change together.



End of Review

Principles of OO Design

- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

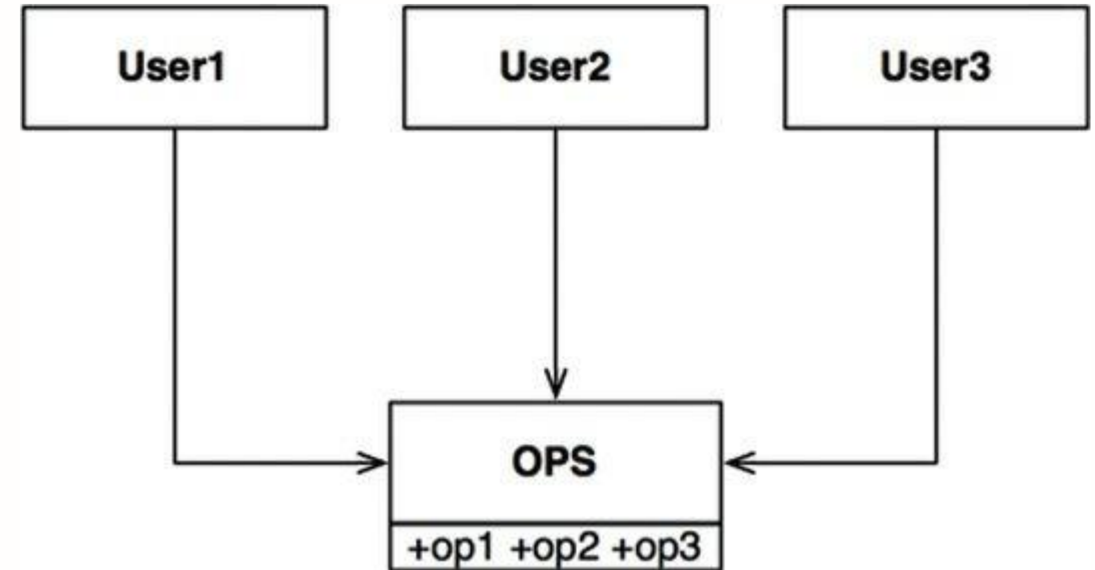
Interface Segregation Principle

Interface Segregation Principle

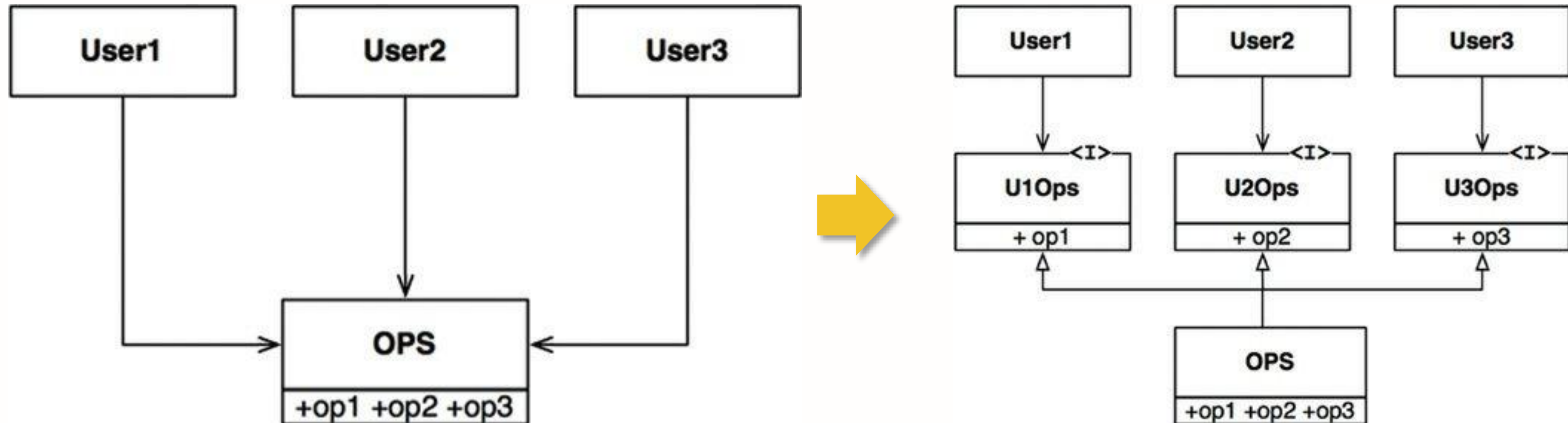
- It is harmful to depend on modules that contain more than you need.
- Clients should not be forced to depend on method that they do not use.
- Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Interface Segregation Principle

- Assume that User1 uses only op1, User2 uses only op2, and User3 uses only op3.
- If OPS is a class written in a language like Java, the source code of User1 will inadvertently depend on op2 and op3, even though it doesn't call them.
- This dependence means that a change to the source code of op2 in OPS will force User1 to be recompiled and redeployed, even though nothing that it cared about has actually changed.

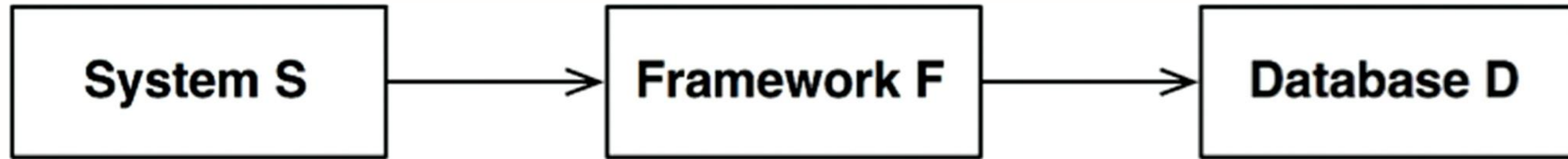


Interface Segregation Principle



Each user only depends on interface that it needs.

Interface Segregation Principle



- An architect working on a system, S, wants to include a certain framework, F, into the system. Now suppose that the authors of F have bound it to a particular database, D. So S depends on F. which depends on D.
- Suppose that D contains features that F does not use and, therefore, that S does not care about. Changes to those features within D may well force the redeployment of F and, therefore, the redeployment of S. Even worse, a failure of one of the features within D may cause failures in F and S.

Interface Segregation Principle

```
// interface segregation principle - bad example
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ...... eating in launch break
        }
}

class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}
```

```
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Changes of Requirement:
Some robots came in the company they work as well, but they don't eat so they don't need a launch break.

Robot class **shouldn’t** directly implements IWorker interface. Since it doesn’t need lunch break. **If Robot class implements IWorker, it violates ISP**.

Interface Segregation Principle

```
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
	public void work();
}

interface IFeedable{
	public void eat();
}

class Worker implements IWorkable, IFeedable{
	public void work() {
		// ....working
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Robot implements IWorkable{
	public void work() {
		// ....working
	}
}
```

```
class SuperWorker implements IWorkable, IFeedable{
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Manager {
	Workable worker;

	public void setWorker(Workable w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

IWorker interface is segregated into IFeedable and IWorkable. **It is an implementation of ISP**.

Dependency Inversion Principle

Dependency Inversion Principle

- The most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.
- Abstraction (Interfaces) are less volatile than concretions (implementations).
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- It is the *volatile* concrete elements of our system that we want to avoid depending on. Those are the modules that we are actively developing, and that are undergoing frequent change.

Dependency Inversion Principle

```
// Dependency Inversion Principle - Bad example

class Worker {

        public void work() {

                // ....working

        }

}
```

```
class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

```
class Manager {

        Worker worker;


        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

Changes of Requirement:
We need to add a new module (SuperWorker) to our application to model the changes in the company structure determined by the employment of new specialized workers. **If Manager class depends on Worker class, we have to change Manager class.**

Dependency Inversion Principle

```
// Dependency Inversion Principle - Good example
interface IWorker {
	public void work();
}

class Worker implements IWorker{
	public void work() {
		// ....working
	}
}

class SuperWorker  implements IWorker{
	public void work() {
		//.... working much more
	}
}
```

```
class Manager {
	IWorker worker;

	public void setWorker(IWorker w) {
		worker = w;
	}

	public void manage() {
		worker.work();
	}
}
```

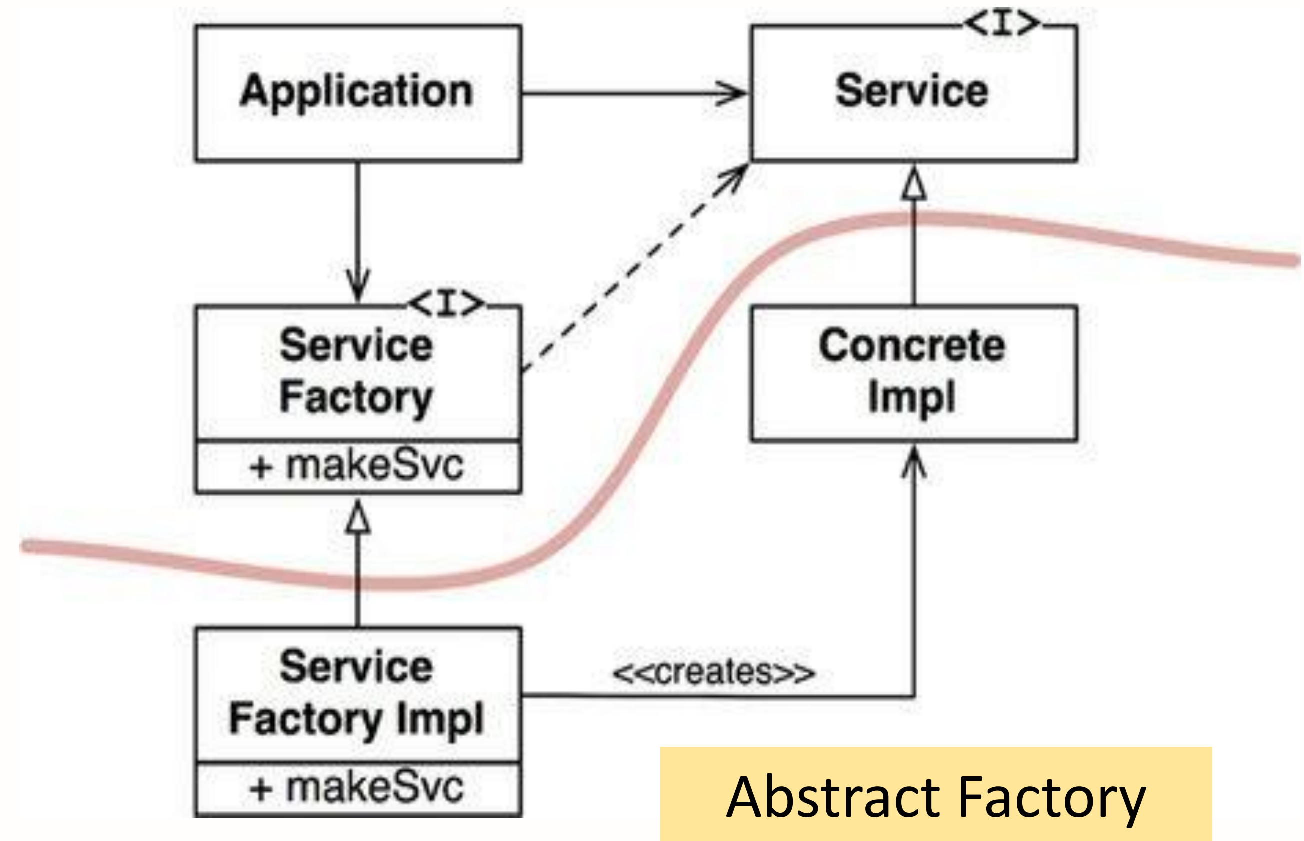
If Manager class depends on IWorker interface, it is easier to accommodate changes of requirement (add SuperWorker class) without changing Manager class. As long as SuperWorker class implements Iworker interface.

Dependency Inversion Principle

- Recommended coding practices:
  - Don’t refer to volatile concrete classes.
  - Don’t derive from volatile concrete classes
  - Don’t override concrete functions
  - Never mention the name of anything concrete and volatile.
- DIP violations cannot be entirely removed, but they can be gathered into a small number of concrete components and kept separate from the rest of the system.

Dependency Inversion Principle

- The curved is an architectural boundary which separates the abstract from the concrete.
- All source code dependencies cross that curved line pointing toward the abstract side.
- Flow of control crosses the curved line in the opposite direction of the source code dependencies. The source code dependencies are inverted against the flow of control



Abstract Factory

References

- Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Pearson. Pearson. 2002
- Martin, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson. 2017.
- https://www.oodesign.com/interface-segregation-principle.html
- https://www.oodesign.com/dependency-inversion-principle.html

bridge to the future
http://www.eepis-its.edu