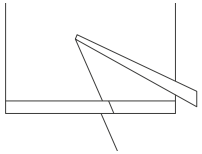
Bring ideas to life
VIA University College



SOLID – Design Principles 2

SWE 1

SOLID

Single Responsibility Principle (SRP)
- A class should **have one, and only one**, reason to change

Open Close Principle (OCP)
- You should be able to extend a classes behaviour, without modifying it

Liskov Substitution Principle (LSP)
- Derived classes must be substitutables for their base classes

Interface Segregation Principle (ISP)
- Make fine grained interfaces that are client specific

Dependency Inversion Principle (DIP)
- Depend on abstractions, not on concretions

Liskov Substitution Principle (LSP)

Inheritance are widely used in OOD/P

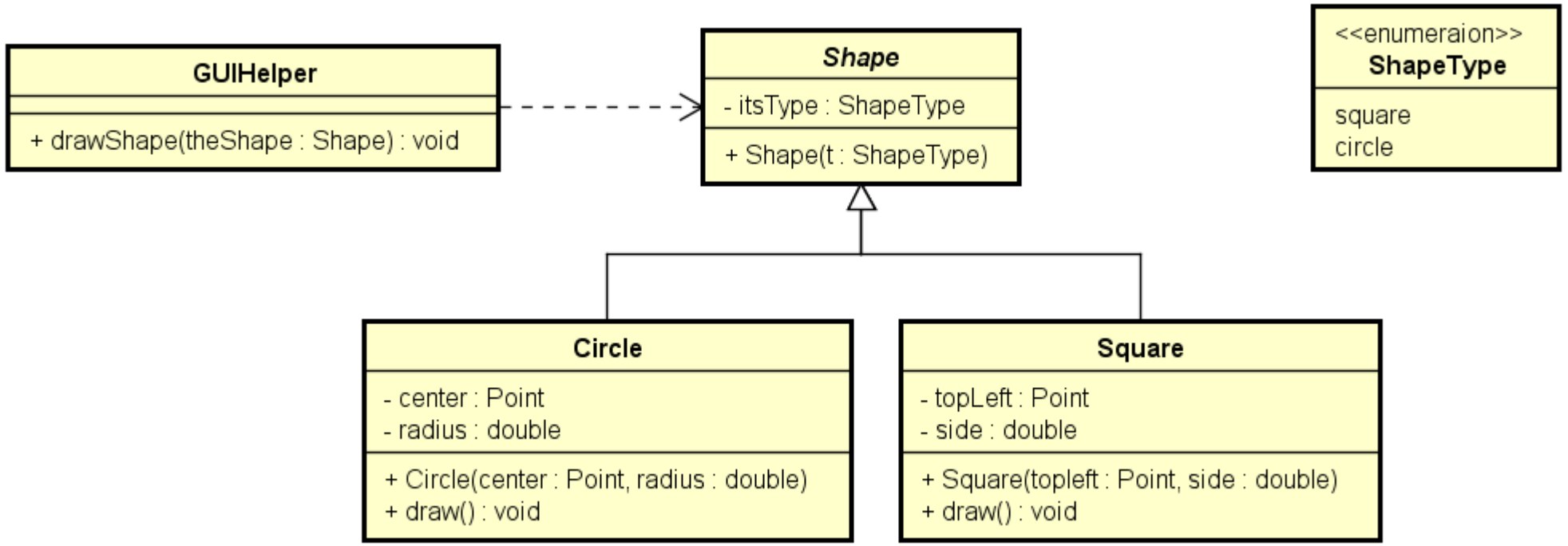
Are there any rules for inheritance?

- "Is-a" rule
- Liskovs rule: "Subtypes must be substitutable for their base types!"

Symptom of LSP Violation: A sub-class does not use/need all attributes/operations in the base-class!!

Liskov Substitution Principle (LSP)

Let's see a violation to better understand it



Circle and Square fulfils the "Is-a" rule

Liskov Substitution Principle (LSP)

```
public abstract class Shape {
    private ShapeType itsType;

    public Shape(ShapeType itsType) {
        this.itsType = itsType;
    }
}
```

Liskov Substitution Principle (LSP)

```
public abstract class Shape {
    private ShapeType itsType;

    public Shape(ShapeType itsType) {
        this.itsType = itsType;
    }
}
```

Liskov Substitution Principle (LSP)

```java
public class Circle extends Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        super(ShapeType.circle);
        this.center = center;
        this.radius = radius;
    }

    public void draw() {
        // Do what is needed to draw a cicle
    }
}
```

```java
public class Square extends Shape {
    private Point center;
    private double side;

    public Square(Point center, double side) {
        super(ShapeType.square);
        this.center = center;
        this.side = side;
    }

    public void draw() {
        // Do what is needed to draw a square
    }
}
```

No big surprises here I hope

Liskov Substitution Principle (LSP)

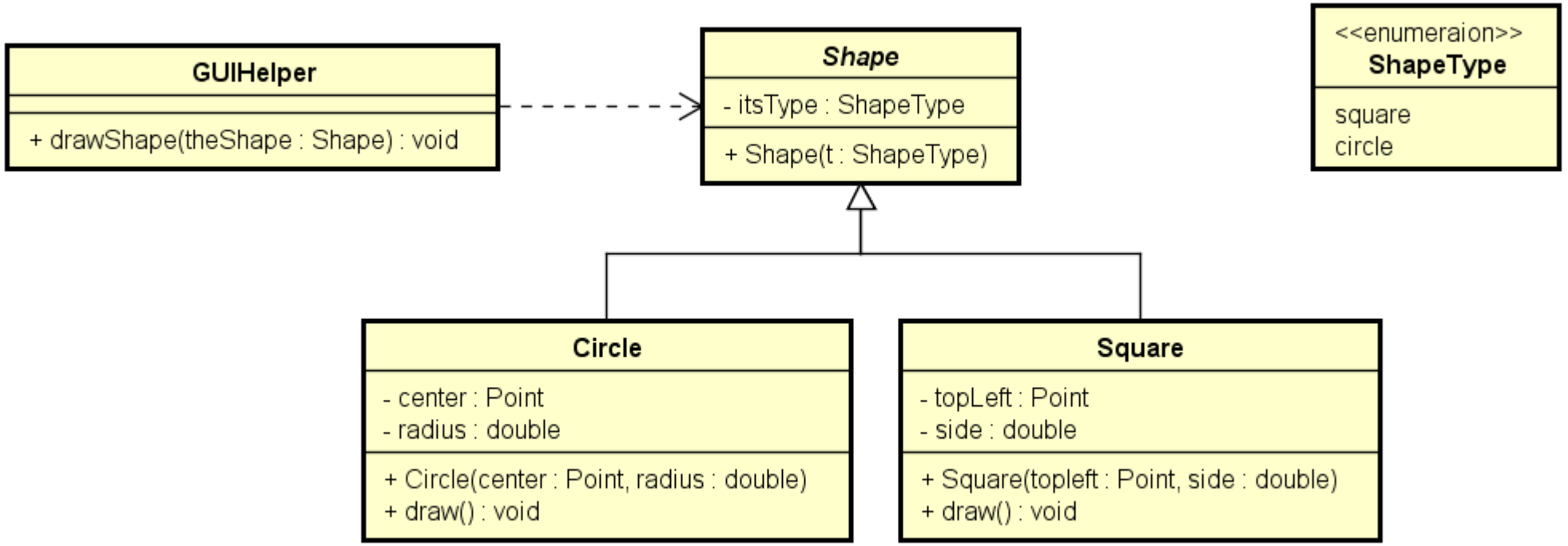
```
public class GUIHelper {
    public void drawShape(Shape shape) {
        if (shape instanceof Circle) {
            ((Circle) shape).draw();
        } else if (shape instanceof Square) {
            ((Square) shape).draw();
        }
    }
}
```

What to say about this?

- It violates **OCP** why? This is often a consequence of a **LSP** violation somewhere else!
- Circle and Square violates **LSP**

Liskov Substitution Principle (LSP)

Why is this violating the LSP?

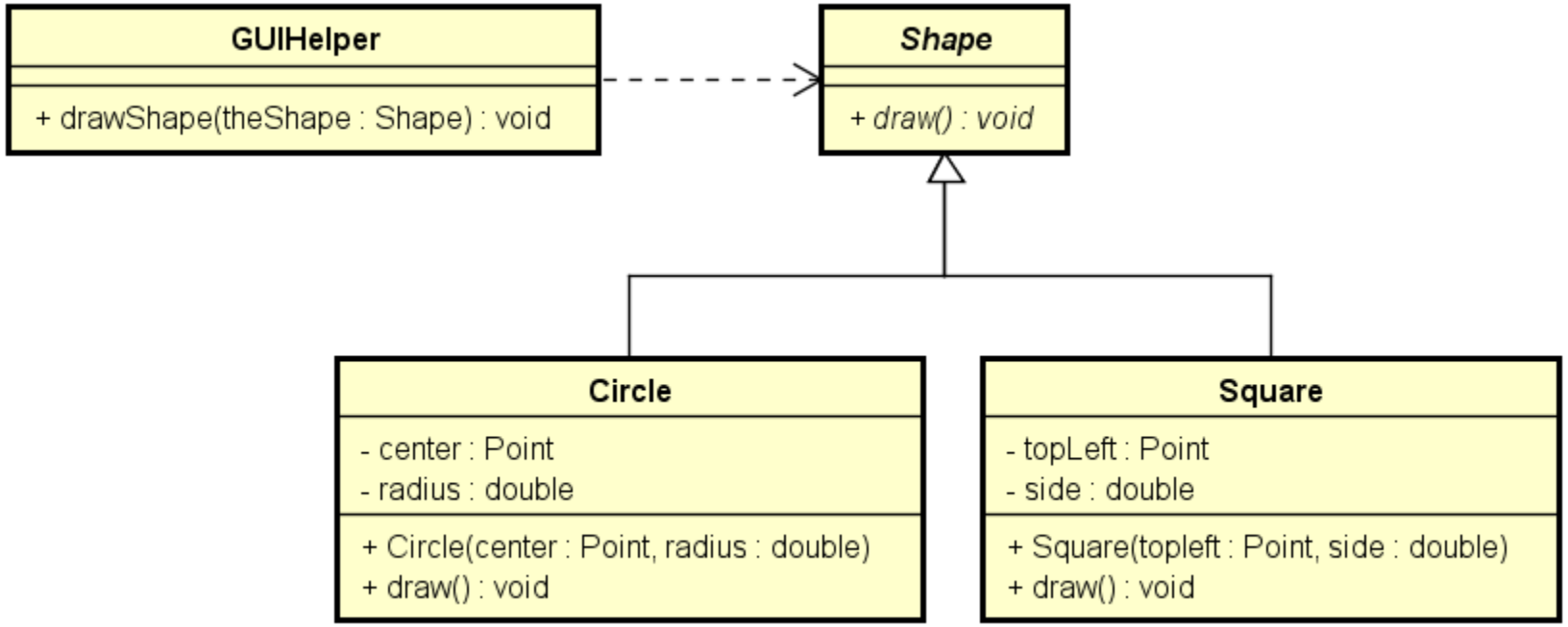


It is not possible to substitute Shape with Circle or Square!

The solution is to make draw() a polymorphic (abstract) method in Shape, and overwrite it in Circle and Square

Liskov Substitution Principle (LSP)

Better Solution:



Interface Segregation Principle (ISP)

```
public abstract class Shape {
    public abstract void draw();
}
```

No need for any ShapeType

```
public class GUIHelper {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}
```

No **OCP** violation in GUIHelper

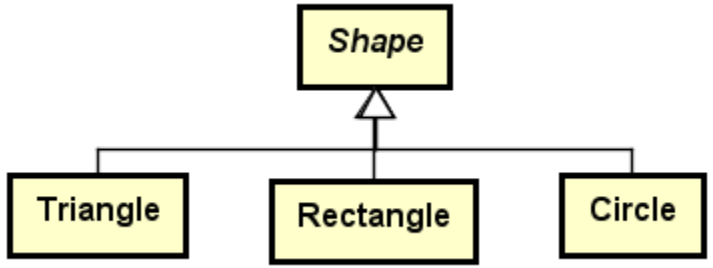
```
public class Circle extends Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public void draw() {
        // Do what is needed to draw a cicle
    }
}
```

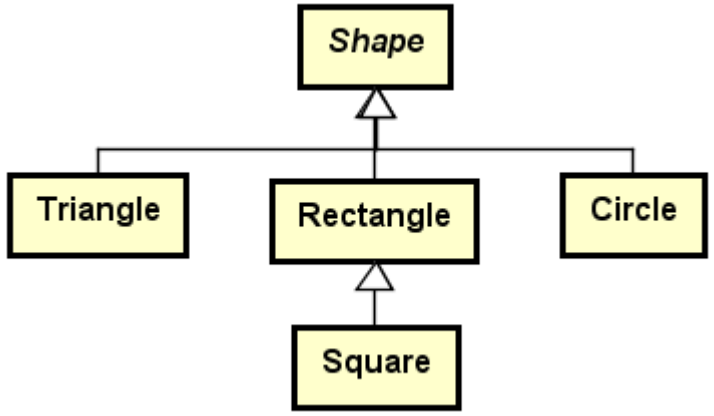
Liskov Substitution Principle (LSP)

The classic example of LSP Violation



Now the Boss wants a Square too

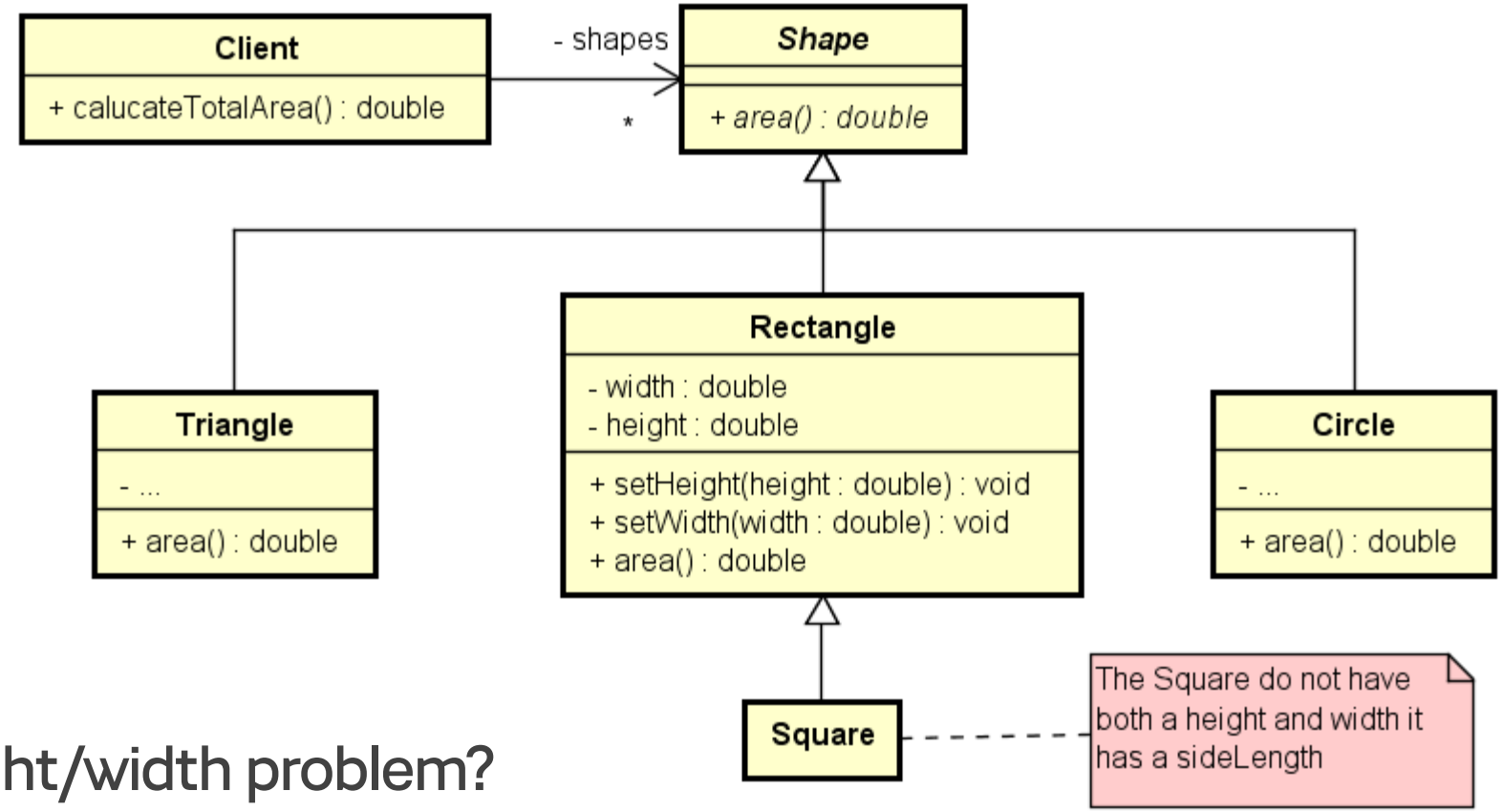
- The Square "Is-a" Rectangle right?



Let's open it a little more up

Liskov Substitution Principle (LSP)

The classic example of LSP Violation



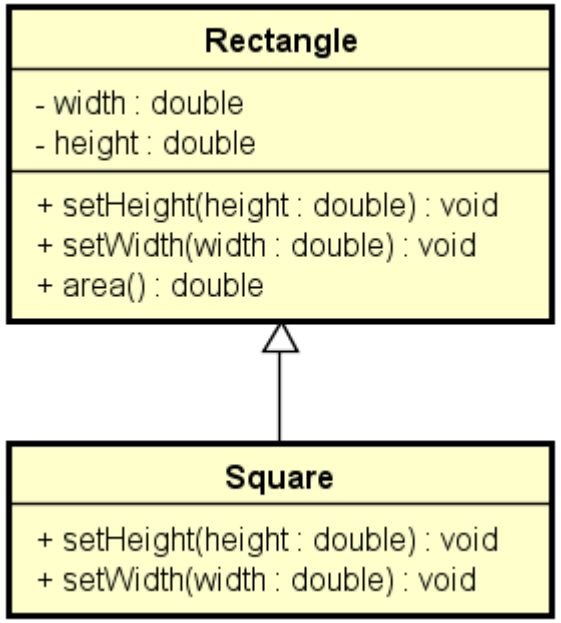
How can Square handle height/width problem?

Liskov Substitution Principle (LSP)

The classic example of LSP Violation

How can Square handle height/width problem?

- Overwrite setHeight and setWidth of course ☺



Liskov Substitution Principle (LSP)

The classic example of LSP Violation

```
public class Rectangle {
    private double height;
    private double width;

    public void setHeight(double height) {
        this.height = height;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double area() {
        return height*width;
    }
}
```

```
public class Square extends Rectangle {
    @Override
    public void setHeight(double height) {
        this.height = height;
        this.width = height;
    }

    @Override
    public void setWidth(double width) {
        this.height = width;
        this.width = width;
    }
}
```

We fixed it – or did we?

Liskov Substitution Principle (LSP)

The classic example of LSP Violation
What is somebody has written a Rectangle test like this?

```
void testArea(Rectangle r) {
    r.setHeight(5.0);
    r.setWidth(6.0);

    assert (30 == r.area()) : "Area test failed";
}
```

When given a Rectangle object as parameter every thing is fine ☺

```
testArea(new Rectangle());
```

But try to give it a Square object as parameter then there is a problem ☹

```
testArea(new Square());
```

```
Exception in thread "main" java.lang.AssertionError: Area test failed
at LSPViolation.Classic.SimpleTest.testArea(SimpleTest.java:9)
at LSPViolation.Classic.SimpleTest.main(SimpleTest.java:15)
```

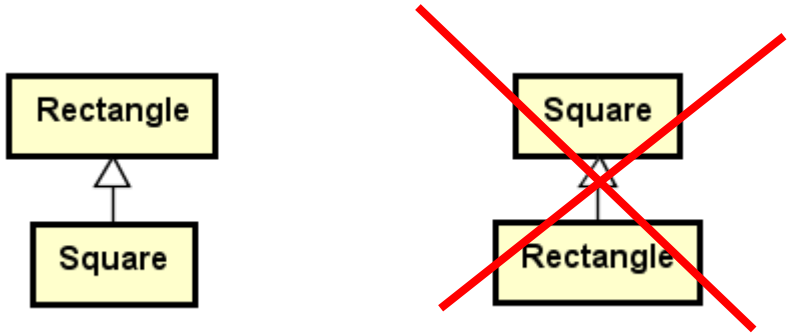
Liskov Substitution Principle (LSP)

The classic example of LSP Violation
What happened???

The Square cannot substitute the Rectangle – so **LSP** is violated!!

Maybe a Square is not a Rectangle, maybe it is just a special kind of Rectangle

We can say that a Square is a Rectangle, but we cannot say that every Rectangle is a Square – That's why it violates **LSP**



Interface-Segregation Principle (ISP)

Clients should not be forced to depend om operations they do not use!

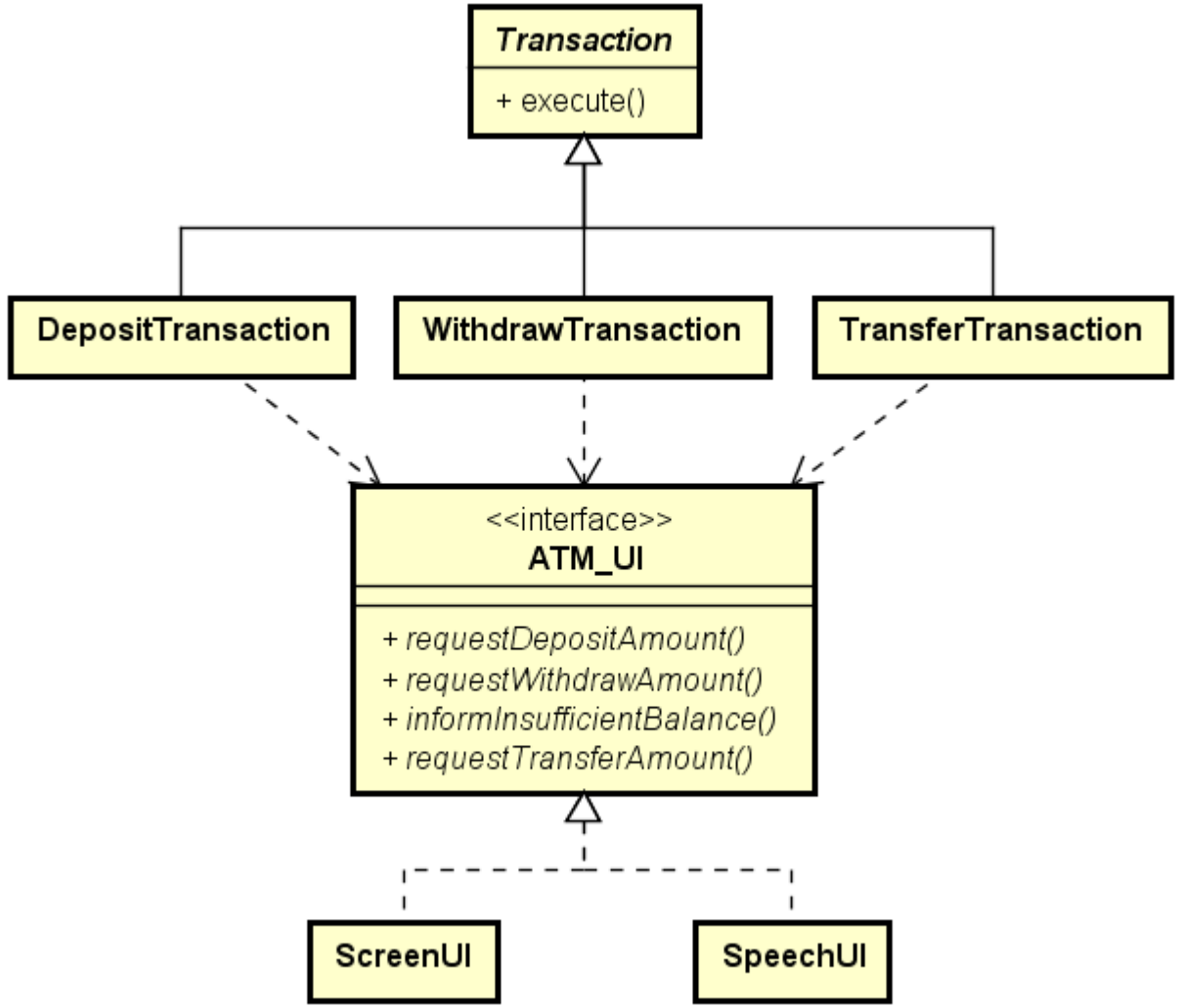
Interfaces to classes should fit the clients needs

Classes with **FAT** interfaces should be broken up into groups that serves different clients

Remember:
What suit one client may not suit another client

Interface-Segregation Principle (ISP)

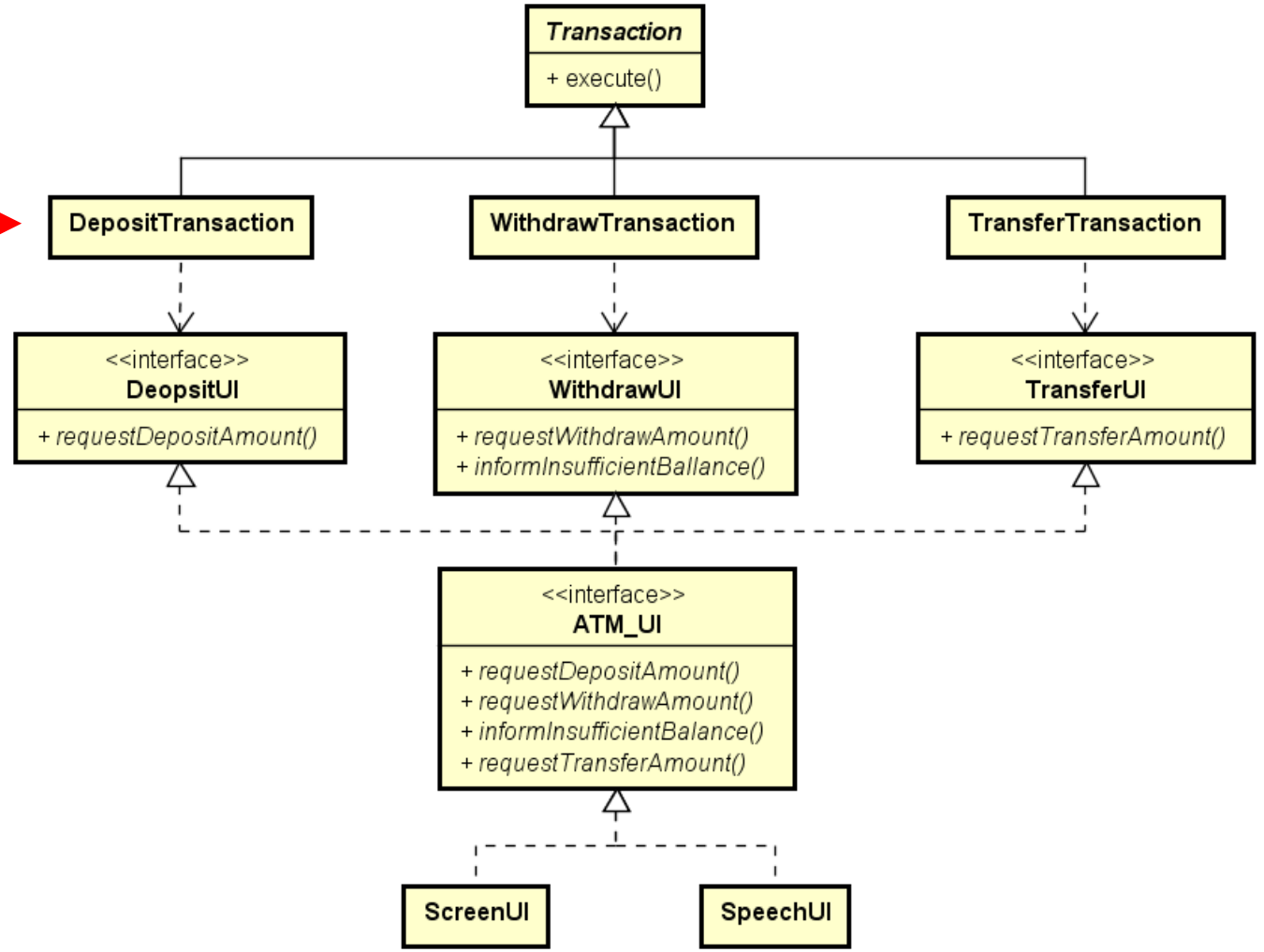
The famous ATM example



Interface-Segregation Principle (ISP)

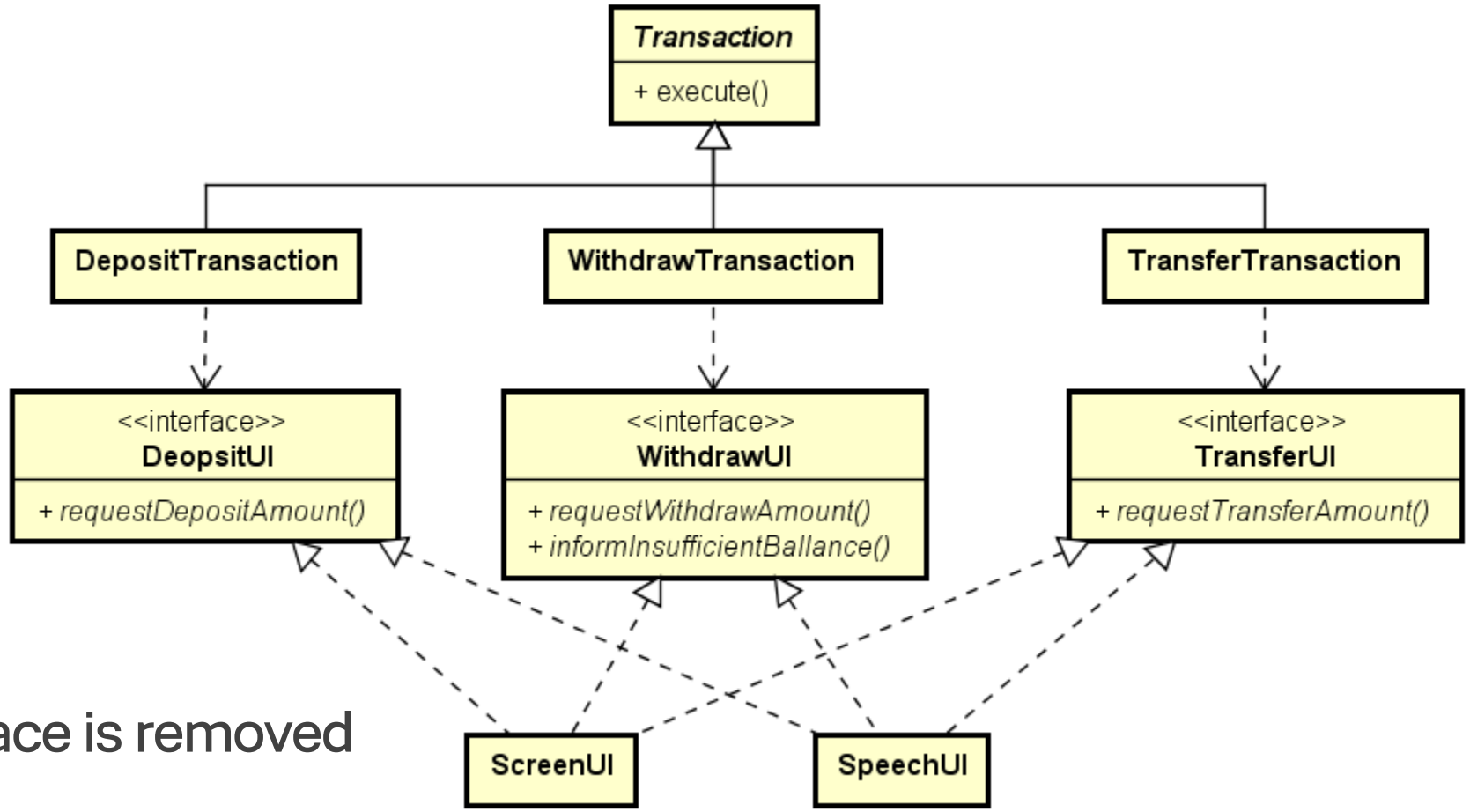
The famous ATM example
Now with ISP

These are only dependent on things they use



Interface-Segregation Principle (ISP)

The famous ATM example
Now with ISP



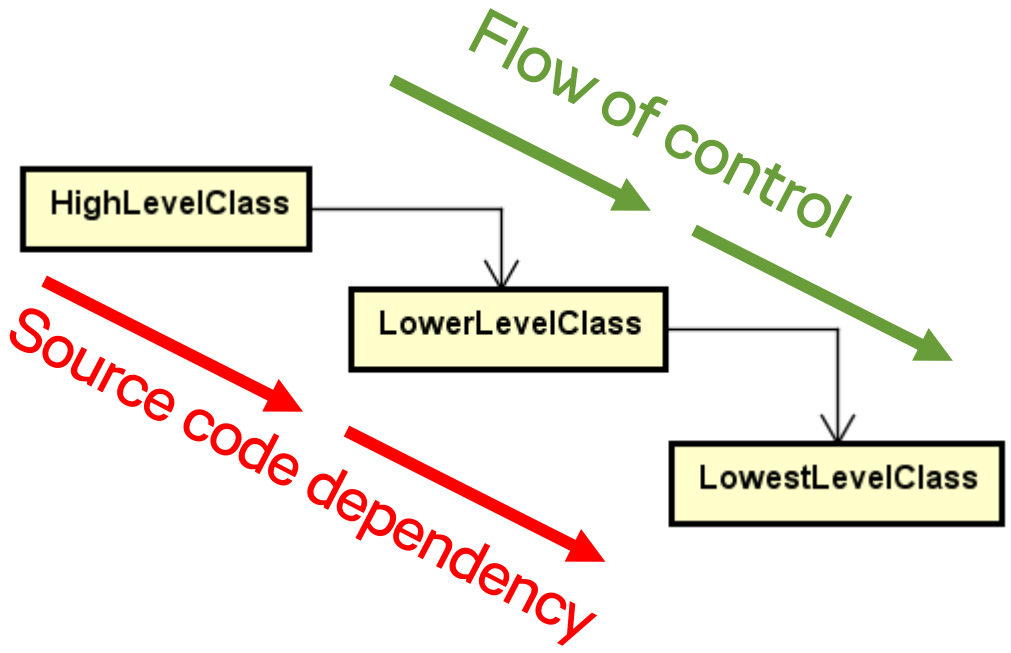
Here the FAT interface is removed

Dependency Inversion Principle (DIP)

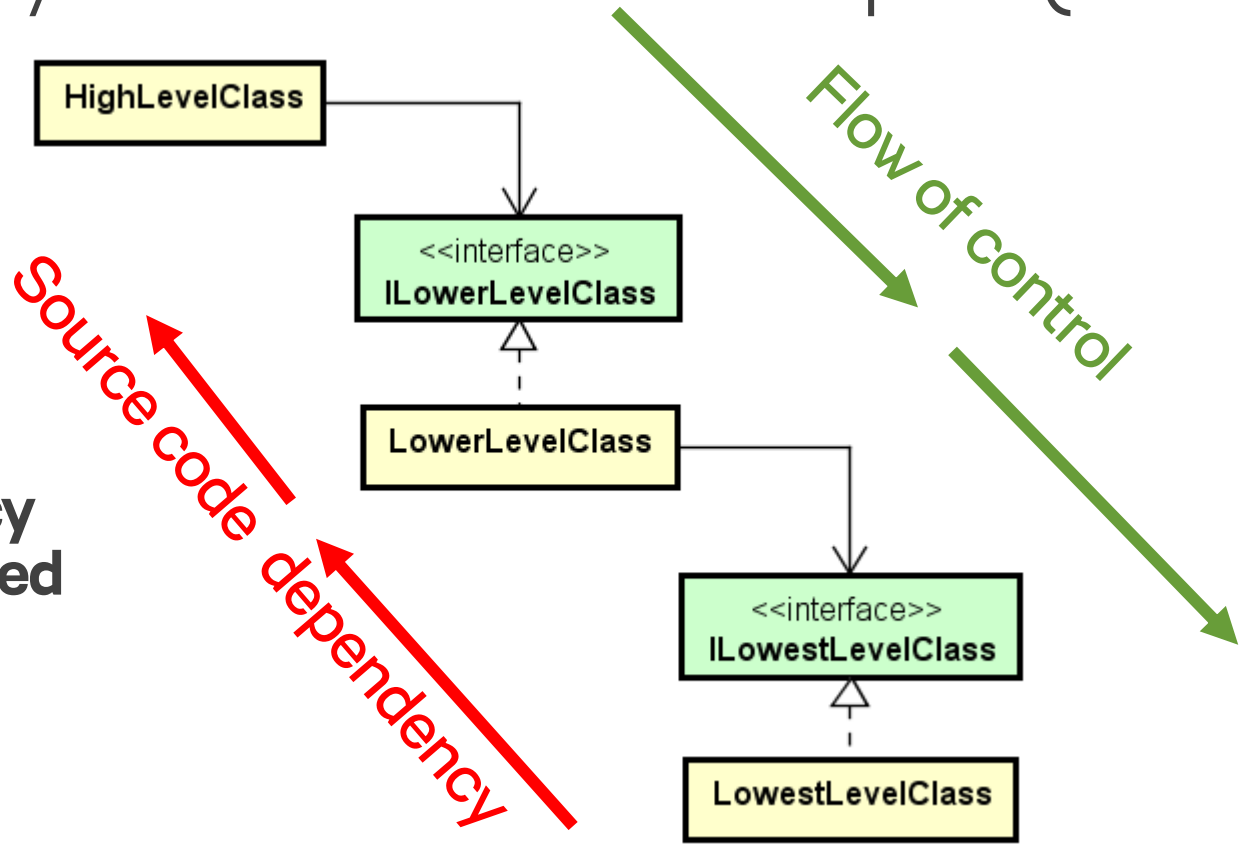
DIP is a strategy saying:

- 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

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)



The dependency has been inverted

- 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

Question to ask yourself about your classes/modules

1. Is it **DRY**?
2. Does it have **one** responsibility?
3. Does everything in it change for the same reason?
4. Does it depend on things that change **less often** than it does?

Correct answers to these questions is YES

If NO you must consider to refactor and change your design/code!!!