Design Principles II

Ergude Bao
Beijing Jiaotong University

Contents

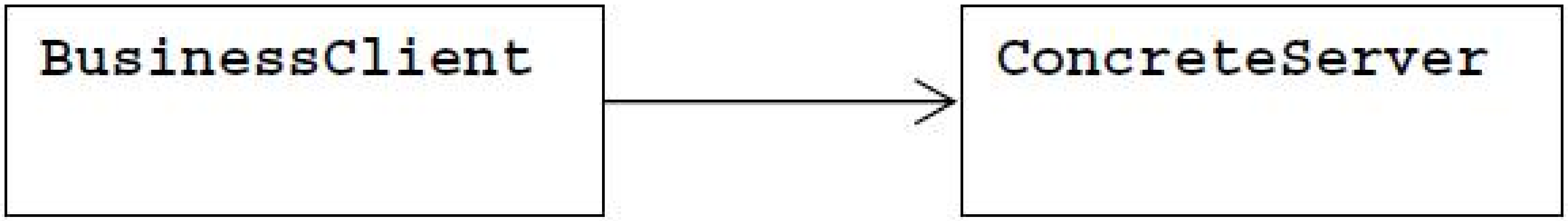
Open-Closed Principle (OCP)

Open-Closed Principle (OCP)

- Open for extension; closed for modification
- What this really means is that you should (re)design so that change leads to extending, not modifying existing code

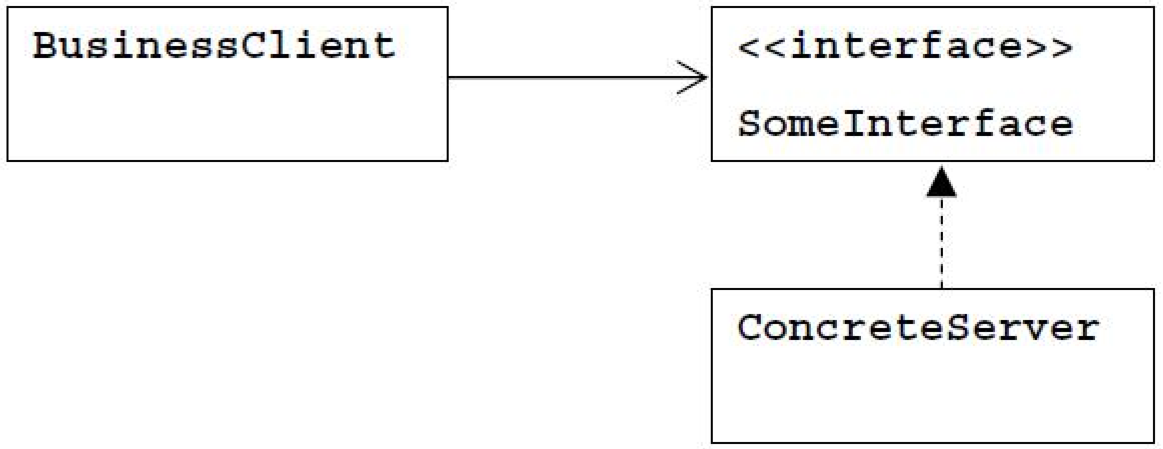
Example I

- If the client has a reference to a concrete server-class, replacing the server leads to modification of the client

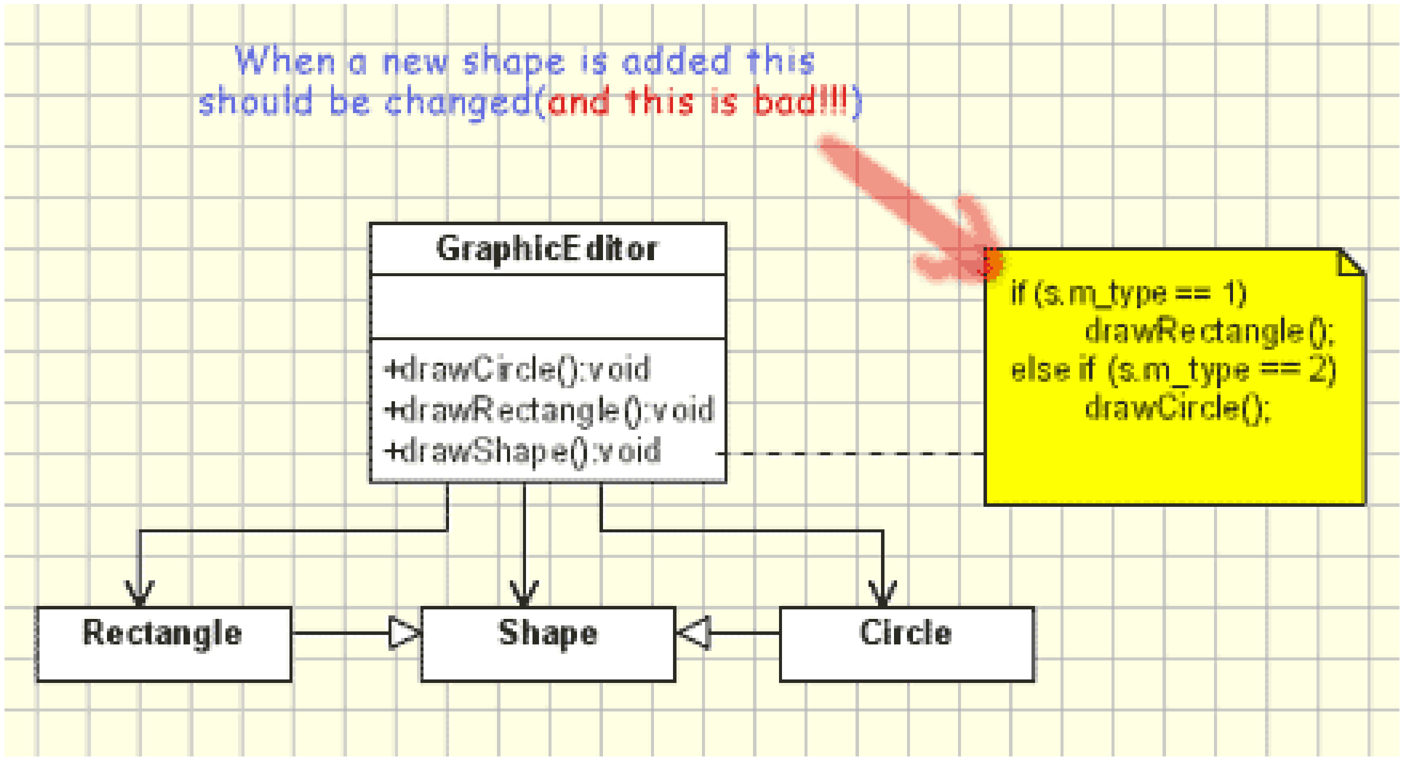


Example I

- If the client has a reference to an interface, replacing the server will not lead to modification of the client. The client still references the interface



Example II



```
if (s.m_type == 1)
        drawRectangle();
else if (s.m_type == 2)
        drawCircle();
```

Example II

```
// 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) {....}
}
```

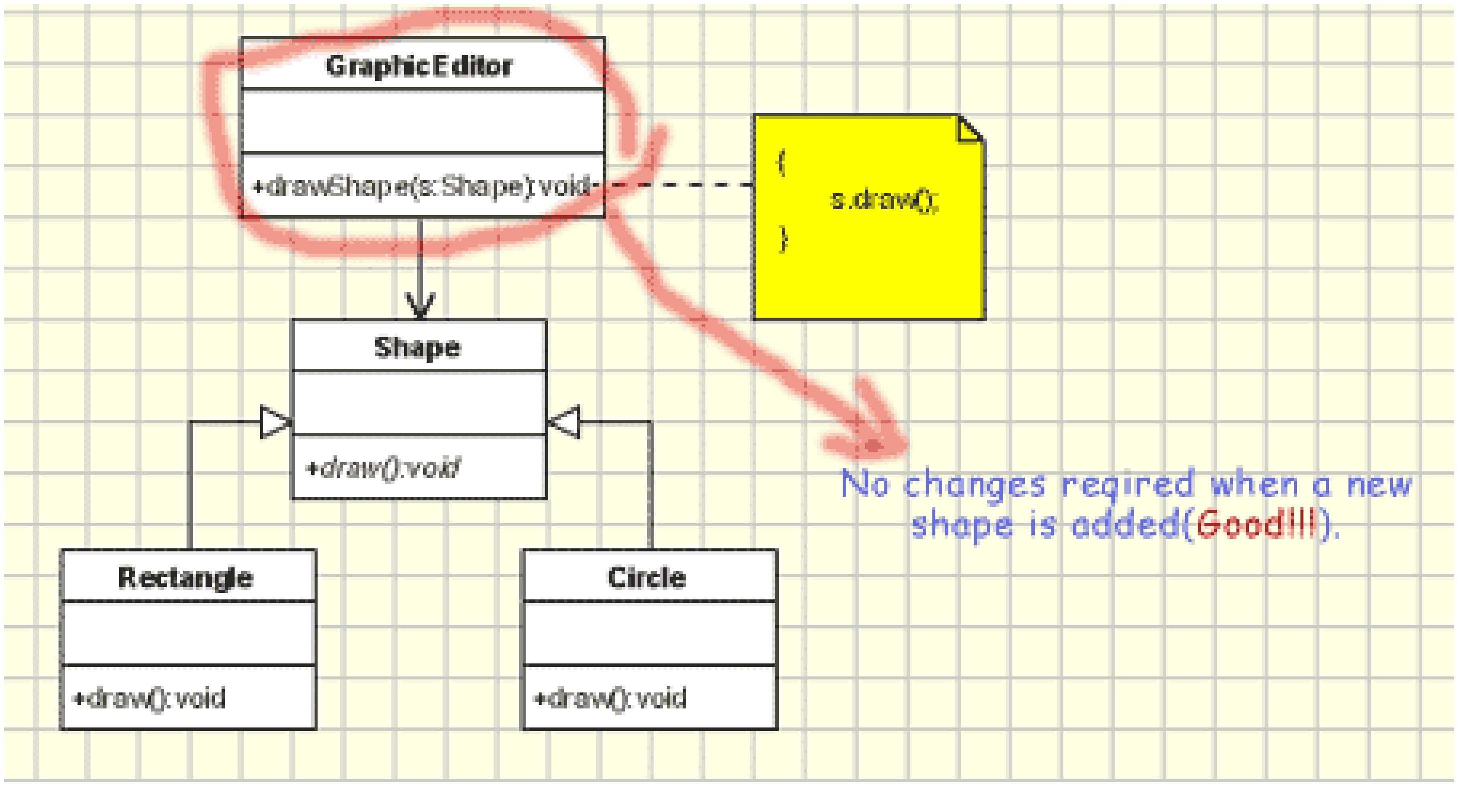
Example II

```
class Shape {
	int m_type;
}

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

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

Example II


GraphicEditor
+drawShape(s:Shape):void
{
s.draw();
}
Shape
+draw():void
Rectangle
+draw():void
Circle
+draw():void
No changes reqired when a new shape is added(Good!!!).

Example II

```
// 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   }
}
```

Example II

- When a new shape is added
  - Open for extension: Add new subclass for new shape
  - Closed for modification: No modification to drawShape()
- What about the smells?
  - Rigidity: Just add new shape classes
  - Fragility: No if’s or switches to maintain
  - Opacity, Needless Repetition, Immobility, … No problem!

Key of OCP

- Find an abstraction for what is common in the variations
- Use polymorphism for what is various in the variations
- Implemented with inheritance

Handling Different Possible Changes

- No matter how “closed” a module is, there will always be some kind of change against which it is not closed
  - What if a new requirement states that the shapes must be drawn in some sorted order, e.g. all Circles must be drawn before all Rectangles?

Handling Different Possible Changes

- Strategy 1: Choose the kinds of changes against which to close the design. Which changes are more likely?
  - Plan for OCP, but wait until the change happens!
- Strategy 2: Stimulate the changes
  - Use short developing cycles
  - Develop the most important features first
  - Write tests first
  - Frequently show those features to stakeholders
  - Release the software early and often

Summary

- OCP
  - Open for extension
  - Closed for modification
- Design for OCP
  - Abstraction
  - Polymorphism
- Not always guaranteed for OCP, since not all changes are predictable
  - Strategic choices
  - From small design to large

Questions

- Please give an example that violates OCP and explain why? How to modify it to conform to OCP?
- How do we measure the quality of inheritance?

Liskov Substitution Principle (LSP)

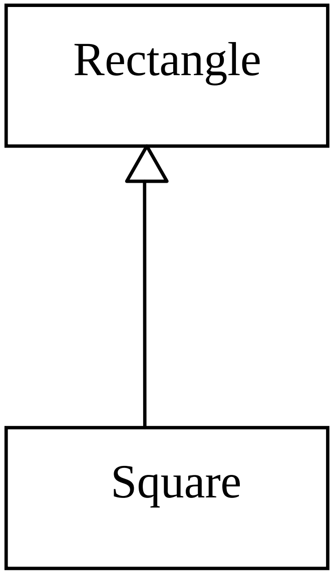
Liskov Substitution Principle (LSP)

- In a computer program, if S is a subclass of T, then objects of class T may be replaced with objects of class S (i.e., objects of class S may be *substituted* for objects of class T) without altering any of the desirable properties of that program (correctness, task performed, etc.)
  - Barbara Liskov (Professor at MIT) proposed in 1988
- Subclasses must be substitutable for their base classes

Example

```
class Rectangle {
       protected int m_width;
       protected int m_height;
       public void setWidth(int width){ m_width=width; }
      public void setHeight(int height){ m_height = height; }
       public int getWidth(){ return m_width; }
       public int getHeight(){ return m_height; }
       public int getArea(){ return m_width * m_height; }
}
```

Example



```
// Bad example
class Square extends Rectangle {
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }
    public void setHeight(int height){
        m_width = height;
        m_height = height;
    }
}
```

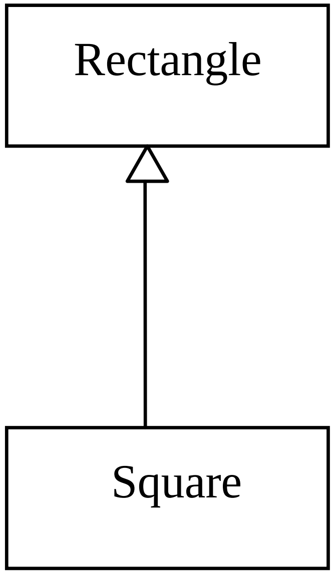
Example

```
class LspTest {
        private static Rectangle getNewRectangle() {
                return new Square();
        }
        public static void main (String args[]) {
                Rectangle r = LspTest.getNewRectangle();
                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle
                System.out.println(r.getArea());
                //
                //
        }
}
```

Example

```
class LspTest {
        private static Rectangle getNewRectangle() {
                return new Square();
        }
        public static void main (String args[]) {
                Rectangle r = LspTest.getNewRectangle();
                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle
                System.out.println(r.getArea());
                // now he's surprised to see that the area is 100 instead of
                // 50
        }
}
```

Example



```
// Good example
class Square extends
    Rectangle {
    protected int edge;
    public void setEdge(int
    edge){
          m_width = edge;
          m_height = edge;
    }
}
```

Key of LSP

- Add new functions to base classes
- Only override abstract functions in base classes

Questions

- Please give an example that violates LSP and explain why? How to modify it to conform to LSP?