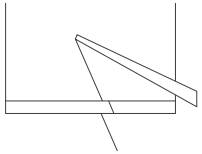
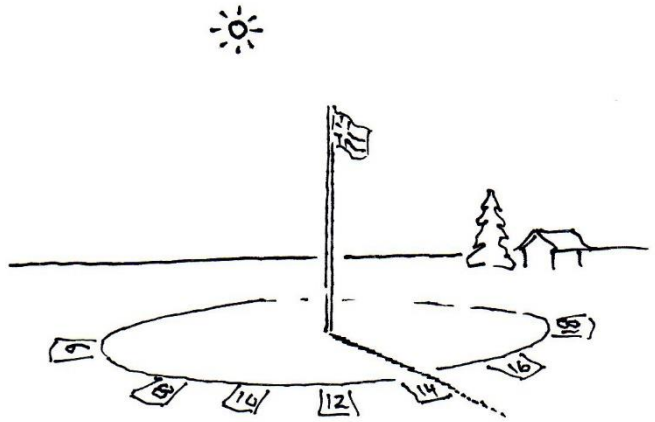
Bring ideas to life
VIA University College



SOLID – Design Principles 1

SWE 1

Motivation



Do you want your application to be a success?

If so – It will change over time!

You think you know how your software will be used in the future

You don't, so make it changeable!!

What is good code/design?

Nearly everything we do in **Design is all about Dependencies!!**

Dependencies makes code fragile!!

- If you refer to something -> you depend on it!
- When the thing you depend on change -> you must change!

To avoid dependencies your design/code should be

- Loosely coupled
  - Inject dependencies
- Highly cohesive
  - Single responsibility (**S**OLID)
- Easily composable
- Context independent

} Must be possible to rearrange to make new behaviour without changing actual code

What the SOLID principles is all about!

Code Smells!!

We start a project with a clear picture of what we want the system to do
The system design is an image in our mind

If we are lucky the "system design" makes it to the first release

Then something goes wrong – customers wants changes
The software starts to rot over time – like bad meat

As time goes by – the rot spreads and grows
making it harder and harder to maintain the code

Even making the smallest and simplest change becomes difficult

Design Smells - Symptom of Poor Design!

1. **Rigidity** – The design is hard to change
2. **Fragility** – The design is easy to break
3. **Immobility** – The design is difficult to reuse
4. **Viscosity** – It is hard to do the right thing
5. **Needless Complexity** – Overdesigned
6. **Needless Repetition** – Mouse abuse
7. **Opacity** – Disorganised expression

Source: Robert C. Martin "Agile Software Development Principles, Patterns and Practices", 2014

Rigidity - Symptom of Poor Design!

Software is difficult to change

- Even small changes leads to a cascade of changes in many **dependent** parts of the software

- Boss: "Why where your estimate for the change so wrong??"

  "It was a lot more complicated then I assumed"

Fragility - Symptom of Poor Design!

Software is difficult to change

- Even small changes leads to a changes and faults in parts of the software that has **no conceptual relationship** with the changed module

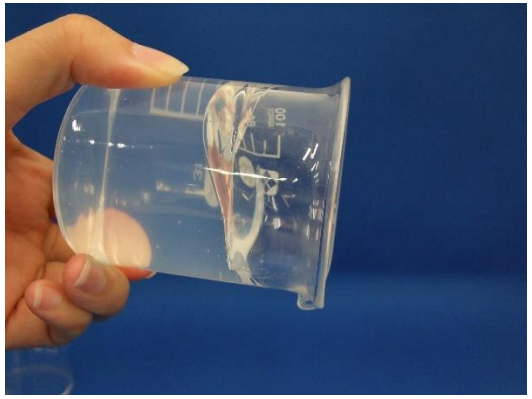
- These are the modules that constantly needs repair, they are always on the bug-list

Immobility - Symptom of Poor Design!

Software parts is difficult to reuse

- The effort and risk involved in separating the reusable modules from the system are too great

Viscosity - Symptom of Poor Design!



Software Viscosity - normally more than one way to do a change

- Some ways preserve the design
- Others do not => Hacks!!

If the design preserving ways are harder to do – then the software viscosity of the design is high

Environment Viscosity

- If the development environment is slow and inefficient
- Source code control system takes hours to commit a few files

- Developers are tempered to make changes that do not preserve the design

Needless Complexity - Symptom of Poor Design!

Software contains elements that aren't currently useful

- Developers anticipates changes to requirements, and put facilities in to deal with future changes

Only implement what is **needed right now**!!

Needless Repetition - Symptom of Poor Design!

Avoid copy and paste coding

Remember the DRY-rule (Don't repeat yourself)

Opacity- Symptom of Poor Design!

The tendency for a module to be difficult to understand

- Code can be written in a clear and understandable way
- Or it can be written in an opaque way

"It was hard to write so it must be hard to read" programming

Always put yourself in the readers shoes when you write code!!

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

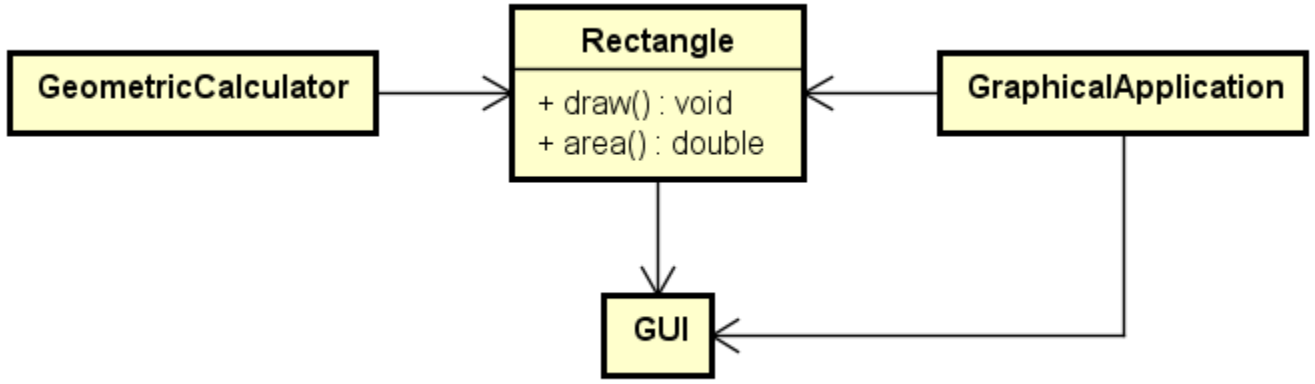
Dependency Inversion Principle (DIP)
- Depend on abstractions, not on concretions

Single Responsibility Principle (SRP)

SRP is a coding strategy about **cohesion**

- A class should have only one reason to change

Is something wrong with this design?



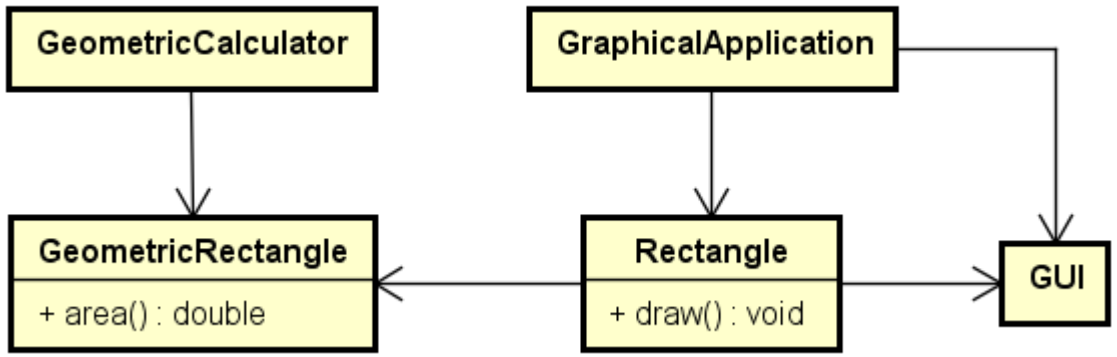
Rectangle has more than one responsibility

Single Responsibility Principle (SRP)

SRP is a coding strategy about **cohesion**

- A class should have only one reason to change

Separated Responsibilities

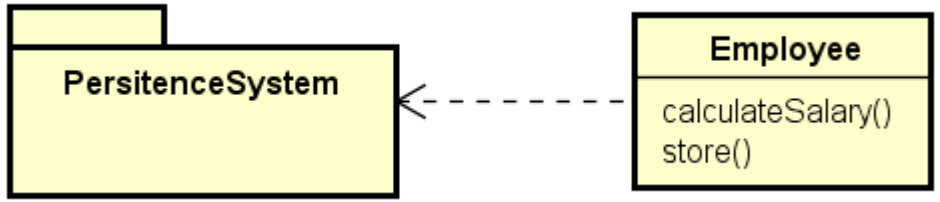


Single Responsibility Principle (SRP)

SRP is a coding strategy about **cohesion**

- A class should have only one reason to change

Persistence leads often to SRP violation



Can be handle using FACADE or PROXY Design Patterns

Open Close Principle (OCP)

OCP is a goal

- Software modules should be open for extension, but closed for modification

Open for extension

- The behaviour of a modules can be extended

Closed for modification

- Extending the module behaviour does not change the source or binary code of the module

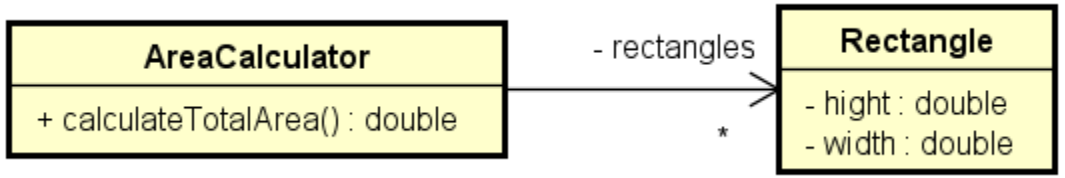
- How can this be possible? – Abstraction is the key!

Open Close Principle (OCP)

What is an abstraction in Java?

- Abstract class, Interface etc.

Is this OK?



How can it be implemented?

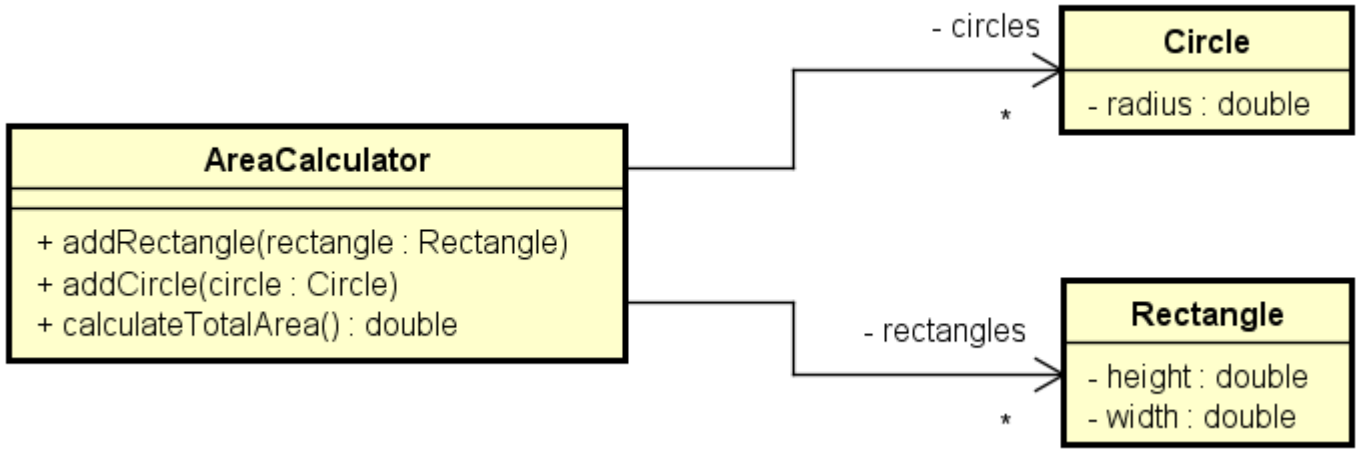
Open Close Principle (OCP)

```
public class AreaCalculator {

   private ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();

    public void addRectangle(Rectangle rectangle) {
      rectangles.add(rectangle);
    }

    public double calculateTotalArea() {
        double totalArea = 0;
        for (Rectangle rectangle : rectangles) {
            totalArea += rectangle.getWidth()*rectangle.getHeight();
        }

      return totalArea;
    }
}
```

Open Close Principle (OCP)

Now the Boss wants Circles too!!

First solution:



How can it be implemented?

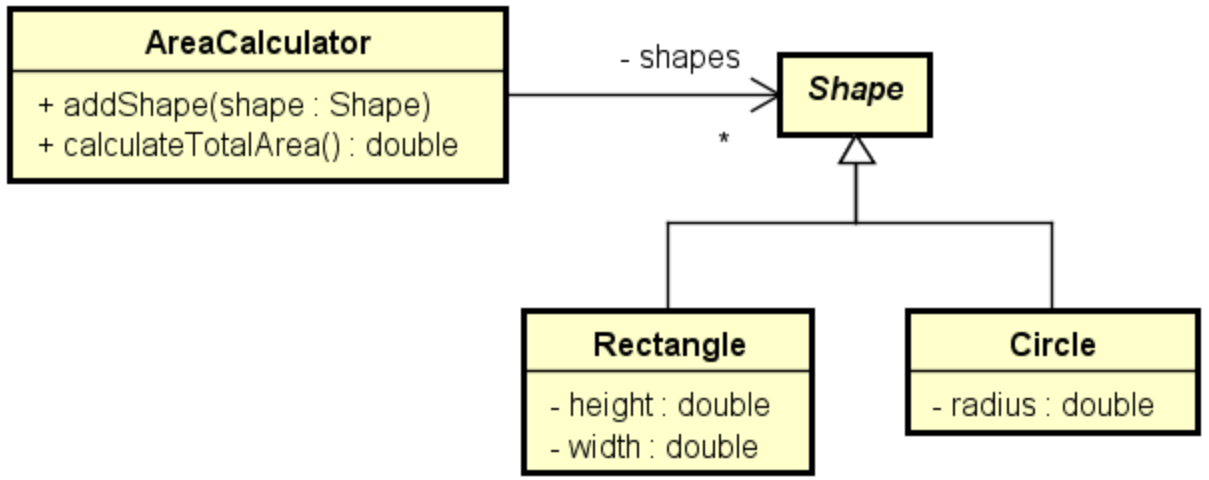
Open Close Principle (OCP)

```
public class AreaCalculator {
     private ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
     private ArrayList<Circle> circles = new ArrayList<Circle>();

     public void addRectangle(Rectangle rectangle) {
        rectangles.add(rectangle);
     }

     public void addCircle(Circle circle) {
        circles.add(circle);
     }

     public double calculateTotalArea() {
          double totalArea = 0;
          for (Rectangle rectangle : rectangles) {
            totalArea += rectangle.getWidth() * rectangle.getHeight();
          }

          for (Circle circle : circles ) {
            totalArea += circle.getRadius() * circle.getRadius() * Math.PI;
          }

          return totalArea;
     }
}
```

Open Close Principle (OCP)

Now the Boss wants Circles too!!

Second solution:



How can it be implemented?

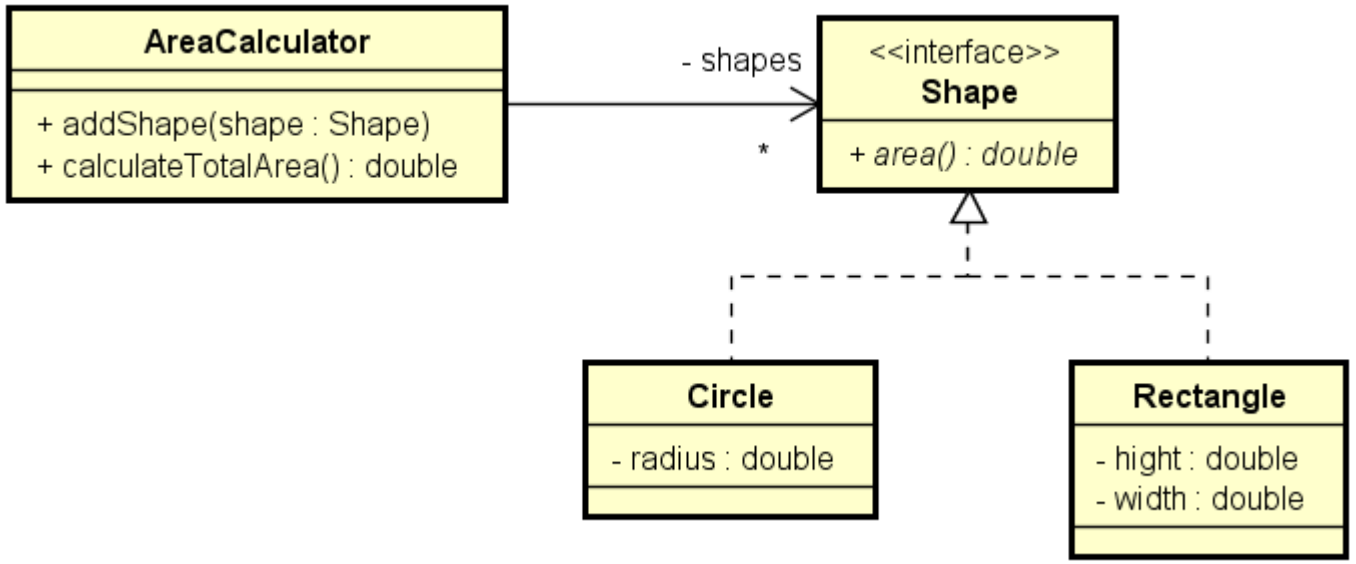
Open Close Principle (OCP)

```
public class AreaCalculator {
     private ArrayList<Shape> shapes = new ArrayList<Shape>();

     public void addShape(Shape shape) {
        shapes.add(shape);
     }

     public double calculateTotalArea() {
          double totalArea = 0;

          for (Shape shape : shapes) {
               if (shape instanceof Rectangle) {
                    Rectangle rectangle = (Rectangle) shape;
                    totalArea += rectangle.getWidth() * rectangle.getHeight();
               } else if (shape instanceof Circle) {
                    Circle circle = (Circle) shape;
                    totalArea += circle.getRadius() * circle.getRadius() * Math.PI;
               }
          }
          return totalArea;
     }
}
```

Open Close Principle (OCP)

Now the Boss wants Circles too!!

Solution conforms to both **OCP** and **DIP** :



How can it be implemented?

Open Close Principle (OCP)

```
public class AreaCalculator {

    private ArrayList<Shape> shapes = new ArrayList<Shape>();

    public void addShape(Shape shape) {
      shapes.add(shape);
    }

    public double calculateTotalArea() {
        double totalArea = 0;

        for (Shape shape : shapes) {
            totalArea += shape.area();
        }

        return totalArea;
    }
}
```

Open Close Principle (OCP)

```
public interface Shape {
    public double area();
}
```

```
public class Circle implements Shape{
     private double radius;

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

     public double getRadius() {
        return radius;
     }

     @Override
     public double area() {
        return radius*radius*Math.PI;
     }
}
```

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

     public Rectangle(double height, double width) {
          this.height = height;
          this.width = width;
     }

     public double getWidth() {
        return width;
     }

     public double getHeight() {
        return height;
     }

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

Open Close Principle (OCP)

One indicator for **OCP** violation is any use of

`instanceof`

```
if (shape instanceof Rectangle) {
```

SOLID

Not SOLID Code

Smells bad

SOLID Code

Smells good

Exercise

1. Take a look on some of your old code or find some on the internet, and see if you can find violations of

Let me know if you can't find some code

- Single Responsibility Principle (SRP)
- Open Close Principle (OCP)

1. Copy the violating code to Power Point
   - Include a Class Diagram that shows the violation

2. Refactor the code to fulfil SRP and OCP

3. Copy the refactored code to Power Point
   - Include a Class Diagram that shows the solution

4. Present the Power Point to the class