Refactoring

Dario Campagna

Why refactoring?

Clean code

We want code that's easy to understand, to evolve, to maintain.

No rotting code

We want to keep the code from becoming rigid, fragile, inseparable, opaque.

Sustain pace

We want to protect us against the long-term erosion of our capacity to deliver features.

Refactoring

Safely improve the design of existing code.

Safely

Take baby steps, keep test bar green.

Improve the design

Does not add new functionalities.

Existing code

It is not rewriting from scratch.

What to **look for** when refactoring?

Simple Design

According to Kent Beck, a design is “simple” if it follows these rules:

1. Runs all the tests
2. Contains no duplication
3. Express the intent of the programmer
4. Minimizes the number of classes and methods

Minimize Duplication

Duplication of knowledge.

```
public class Cylinder {

    private final double radius;
    private final double height;

    public Cylinder(double radius, double height) {
        this.radius = radius;
        this.height = height;
    }

    public double volume() {
        return Math.PI * Math.pow(radius, 2) * height;
    }

    public double surface() {
        return 2 * Math.PI * Math.pow(radius, 2) + 2 * Math.PI * radius * height;
    }
}
```

Minimize Duplication

Duplication of knowledge.

```java
public class Cylinder {

    private final double radius;
    private final double height;

    public Cylinder(double radius, double height) {
        this.radius = radius;
        this.height = height;
    }

    public double volume() {
        return Math.PI * Math.pow(radius, 2) * height;
    }

    public double surface() {
        return 2 * Math.PI * Math.pow(radius, 2) + 2 * Math.PI * radius * height;
    }
}
```

Minimize Duplication

Extract method.

```
public class Cylinder {

    private final double radius;
    private final double height;

    public Cylinder(double radius, double height) {
        this.radius = radius;
        this.height = height;
    }

    public double volume() {
        return baseSurface() * height;
    }

    public double surface() {
        return 2 * baseSurface() + 2 * Math.PI * radius * height;
    }

    private double baseSurface() {
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Minimize Duplication

Extract method.

```
public class Cylinder {

    private final double radius;
    private final double height;

    public Cylinder(double radius, double height) {
        this.radius = radius;
        this.height = height;
    }

    public double volume() {
        return baseSurface() * height;
    }

    public double surface() {
        return 2 * baseSurface() + 2 * Math.PI * radius * height;
    }

    private double baseSurface() {
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Minimize Duplication

Duplication of hard coded data.

```
@Test
public void productNotFound() throws Exception {
    Display display = new Display();
    Sale sale = new Sale(display);

    sale.onBarcode("99999");

    assertEquals("Product not found for 99999", display.getText());
}
```

```
public class Sale {

    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void onBarcode(String barcode) {
        display.setText("Product not found for 99999");
    }
}
```

Minimize Duplication

Duplication of hard coded data.

```java
@Test
public void productNotFound() throws Exception {
    Display display = new Display();
    Sale sale = new Sale(display);

    sale.onBarcode("99999");

    assertEquals("Product not found for 99999", display.getText());
}
```

```java
public class Sale {

    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void onBarcode(String barcode) {
        display.setText("Product not found for 99999");
    }
}
```

Minimize Duplication

Replace literal value with variable.

```
@Test
public void productNotFound() throws Exception {
    Display display = new Display();
    Sale sale = new Sale(display);

    sale.onBarcode("99999");

    assertEquals("Product not found for 99999", display.getText());
}
```

```
public class Sale {

    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void onBarcode(String barcode) {
        display.setText("Product not found for " +
                barcode);
    }
}
```

Minimize Duplication

Replace literal value with variable.

```java
@Test
public void productNotFound() throws Exception {
    Display display = new Display();
    Sale sale = new Sale(display);

    sale.onBarcode("99999");

    assertEquals("Product not found for 99999", display.getText());
}
```

```java
public class Sale {

    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void onBarcode(String barcode) {
        display.setText("Product not found for " +
                barcode);
    }
}
```

Maximize Expressiveness

Method does more than what suggested by its name.

```
private void displayPrice(String barcode) {
    String priceAsText = pricesByBarcode.get(barcode);
    display.setText(priceAsText);
}
```

Maximize Expressiveness

Method does more than what suggested by its name.

```
private void displayPrice(String barcode) {
    String priceAsText = pricesByBarcode.get(barcode);
    display.setText(priceAsText);
}
```

Find

Maximize Expressiveness

Method does more than what suggested by its name.

```java
private void displayPrice(String barcode) {
    String priceAsText = pricesByBarcode.get(barcode);
    display.setText(priceAsText);
}
```

Find

Display

Maximize Expressiveness

Conjunction tells us that method has more than one responsibility.

```
private void findPriceAndDisplayAsText(String barcode) {
    String priceAsText = pricesByBarcode.get(barcode);
    display.setText(priceAsText);
}
```

Maximize Expressiveness

Two methods. Each method has one responsibility.

```
private String findPrice(String barcode) {
    return pricesByBarcode.get(barcode);
}

private void displayPrice(String priceAsText) {
    display.setText(priceAsText);
}
```

Code Smells

A code smell is a surface indication that usually corresponds to a deeper problem in the system.

- Quick to spot
- Don’t always indicate a problem
- Often an indicator of a problem rather than the problem themselves

Code Smells

For example

Long method

Dead Code

Primitive Obsession

…

Resources

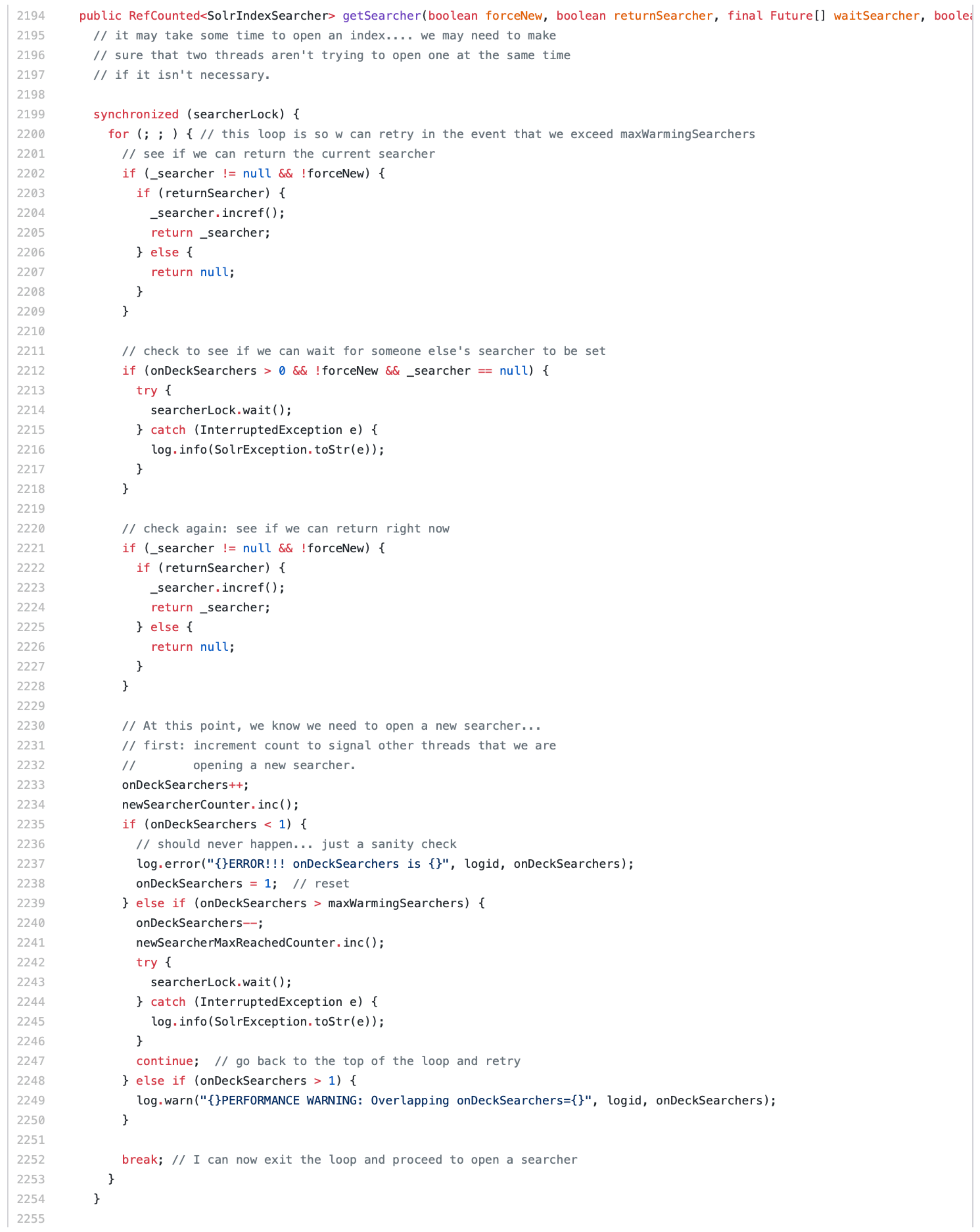
[Refactoring: Improving the Design of Existing Code]

[sourcemaking.com]

Long Method

A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.

- Extract method
- Introduce parameter object
- Decompose conditionals
- …

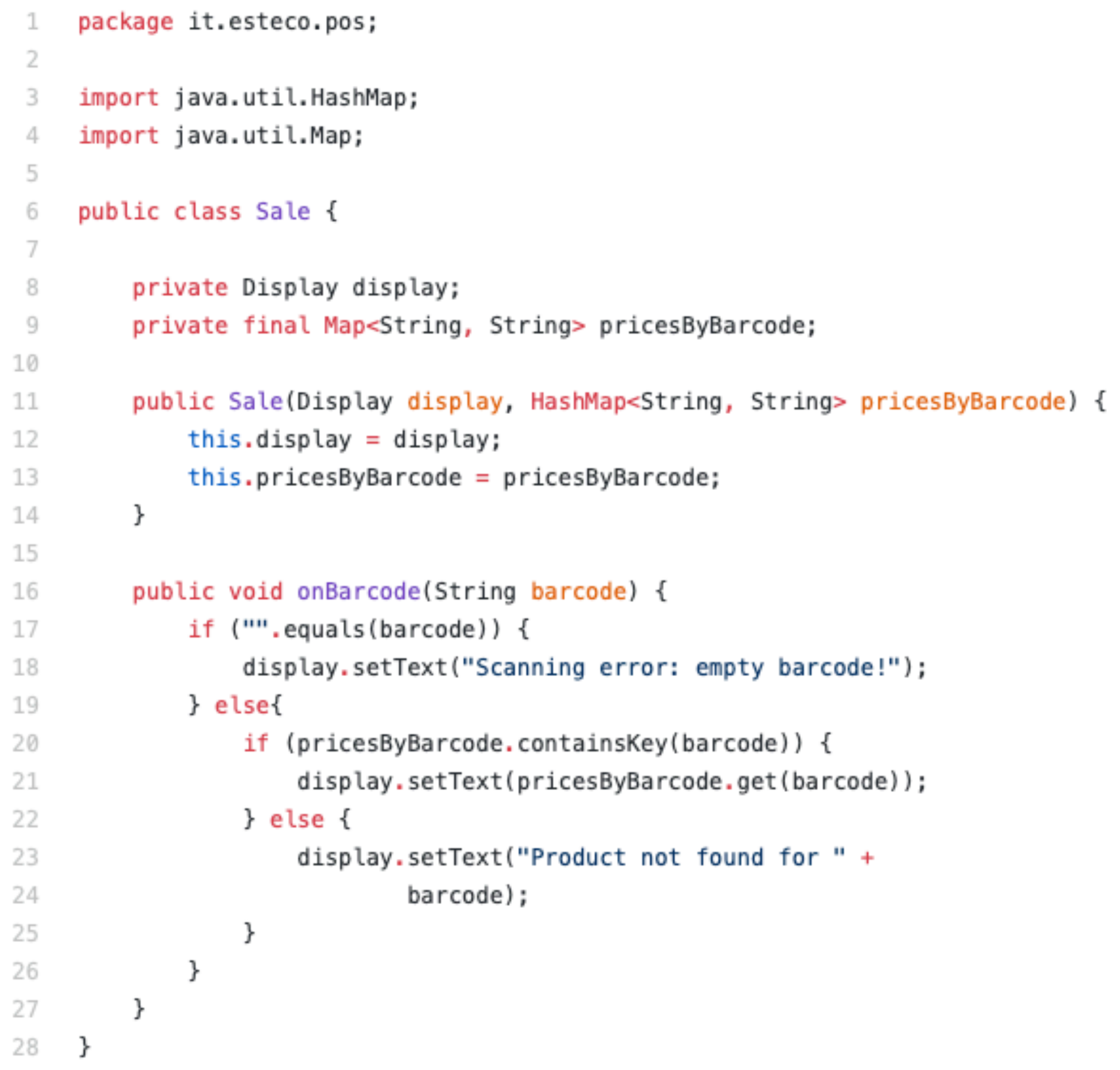


```java
public RefCounted<SolrIndexSearcher> getSearcher(boolean forceNew, boolean returnSearcher, final Future[] waitSearcher, boolea
  // it may take some time to open an index.... we may need to make
  // sure that two threads aren't trying to open one at the same time
  // if it isn't necessary.

  synchronized (searcherLock) {
    for (; ; ) { // this loop is so w can retry in the event that we exceed maxWarmingSearchers
      // see if we can return the current searcher
      if (_searcher != null && !forceNew) {
        if (returnSearcher) {
          _searcher.incref();
          return _searcher;
        } else {
          return null;
        }
      }

      // check to see if we can wait for someone else's searcher to be set
      if (onDeckSearchers > 0 && !forceNew && _searcher == null) {
        try {
          searcherLock.wait();
        } catch (InterruptedException e) {
          log.info(SolrException.toStr(e));
        }
      }

      // check again: see if we can return right now
      if (_searcher != null && !forceNew) {
        if (returnSearcher) {
          _searcher.incref();
          return _searcher;
        } else {
          return null;
        }
      }

      // At this point, we know we need to open a new searcher...
      // first: increment count to signal other threads that we are
      //        opening a new searcher.
      onDeckSearchers++;
      newSearcherCounter.inc();
      if (onDeckSearchers < 1) {
        // should never happen... just a sanity check
        log.error("{}ERROR!!! onDeckSearchers is {}", logid, onDeckSearchers);
        onDeckSearchers = 1;  // reset
      } else if (onDeckSearchers > maxWarmingSearchers) {
        onDeckSearchers--;
        newSearcherMaxReachedCounter.inc();
        try {
          searcherLock.wait();
        } catch (InterruptedException e) {
          log.info(SolrException.toStr(e));
        }
        continue;  // go back to the top of the loop and retry
      } else if (onDeckSearchers > 1) {
        log.warn("{}PERFORMANCE WARNING: Overlapping onDeckSearchers={}", logid, onDeckSearchers);
      }

      break; // I can now exit the loop and proceed to open a searcher
    }
  }
```

Primitive Obsession

Use of primitive types instead of small objects for simple tasks.

- Replace data value with object
- Replace type code with class
- Replace array with object
- …



```java
package it.esteco.pos;

import java.util.HashMap;
import java.util.Map;

public class Sale {

    private Display display;
    private final Map<String, String> pricesByBarcode;

    public Sale(Display display, HashMap<String, String> pricesByBarcode) {
        this.display = display;
        this.pricesByBarcode = pricesByBarcode;
    }

    public void onBarcode(String barcode) {
        if ("".equals(barcode)) {
            display.setText("Scanning error: empty barcode!");
        } else{
            if (pricesByBarcode.containsKey(barcode)) {
                display.setText(pricesByBarcode.get(barcode));
            } else {
                display.setText("Product not found for " +
                        barcode);
            }
        }
    }
}
```

Exercises

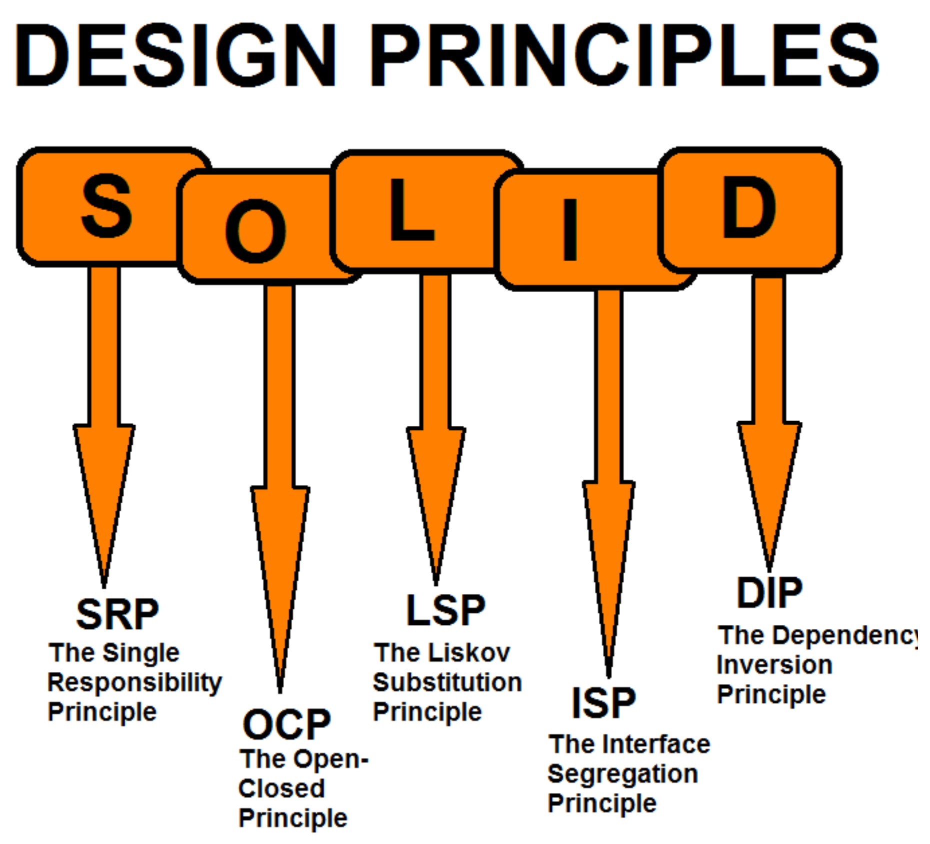
Let’s find some code smells and duplications.

- https://github.com/nerdschoolbergen/code-smells/blob/master/assignment/src/main/java/nerdschool/bar/Pub.java
- https://github.com/nicoleorzan/berlin_clock/blob/master/src/main/java/berlinclock

S.O.L.I.D. Principles

Principles of class design.

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


DESIGN PRINCIPLES
S
O
L
I
D
SRP
The Single Responsibility Principle
OCP
The Open-Closed Principle
LSP
The Liskov Substitution Principle
ISP
The Interface Segregation Principle
DIP
The Dependency Inversion Principle

Single Responsibility Principle

A class should have only one reason to change.

- We want classes to be cohesive
- One and only one responsibility
- Can be applied to methods too

```java
public class Rectangle {

    private double width;
    private double height;
    private Graphics graphics;

    // ...

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

    public void draw() {
        // Do something with Graphics
    }

}
```

Single Responsibility Principle

A class should have only one reason to change.

```
public class Rectangle {

    private double width;
    private double height;
    private Graphics graphics;

    // ...

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

    public void draw() {
        // Do something with Graphics
    }

}
```

- We want classes to be cohesive
- One and only one responsibility
- Can be applied to methods too

Single Responsibility Principle

A class should have only one reason to change.

- We want classes to be cohesive
- One and only one responsibility
- Can be applied to methods too

```
public class Rectangle {

    private double width;
    private double height;
    private Graphics graphics;

    // ...

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

    public void draw() {
        // Do something with Graphics
    }

}
```

Single Responsibility Principle

Move responsibilities to other (new) classes.

- Composition over inheritance
- Related behaviors close to each other

```java
public class GeometricRectangle {

    private double width;
    private double height;

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

}

public class Rectangle {

    private GeometricRectangle geometricRectangle;
    private Graphics graphics;

    // ...

    public void draw() {
        // Draw geometricRectangle using Graphics
    }

}
```

Single Responsibility Principle

Move responsibilities to other (new) classes.

- Composition over inheritance
- Related behaviors close to each other

```
public class GeometricRectangle {

    private double width;
    private double height;

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

}

public class Rectangle {

    private GeometricRectangle geometricRectangle;
    private Graphics graphics;

    // ...

    public void draw() {
        // Draw geometricRectangle using Graphics
    }

}
```

Single Responsibility Principle

Move responsibilities to other (new) classes.

- Composition over inheritance
- Related behaviors close to each other

```java
public class GeometricRectangle {

    private double width;
    private double height;

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

}

public class Rectangle {

    private GeometricRectangle geometricRectangle;
    private Graphics graphics;

    // ...

    public void draw() {
        // Draw geometricRectangle using Graphics
    }

}
```

Single Responsibility Principle

Move responsibilities to other (new) classes.

- Composition over inheritance
- Related behaviors close to each other

```java
public class GeometricRectangle {

    private double width;
    private double height;

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

}

public class Rectangle {

    private GeometricRectangle geometricRectangle;
    private Graphics graphics;

    // ...

    public void draw() {
        // Draw geometricRectangle using Graphics
    }

}
```

Open Closed Principle

Software entities should be open for extension, but closed for modification.

- Minimize changes to existing code when adding new behavior
- Take advantage of object composition and polymorphism

```java
public class Shape {
    // ...
}

public class Rectangle extends Shape {
    // ...
}

public class Circle extends Shape {
    // ...
}

public class GraphicEditor {

    public void drawShape(Shape s) {
        if (s instanceof Rectangle) {
            drawRectangle((Rectangle) s);
        } else if (s instanceof Circle) {
            drawCircle((Circle) s);
        }
    }

    public void drawRectangle(Rectangle rectangle) {
        // ...
    }

    public void drawCircle(Circle c) {
        // ...
    }

}
```

Open Closed Principle

Software entities should be open for extension, but closed for modification.

- Minimize changes to existing code when adding new behavior
- Take advantage of object composition and polymorphism

```java
public class Shape {
    // ...
}

public class Rectangle extends Shape {
    // ...
}

public class Circle extends Shape {
    // ...
}

public class GraphicEditor {

    public void drawShape(Shape s) {
        if (s instanceof Rectangle) {
            drawRectangle((Rectangle) s);
        } else if (s instanceof Circle) {
            drawCircle((Circle) s);
        }
    }

    public void drawRectangle(Rectangle rectangle) {
        // ...
    }

    public void drawCircle(Circle c) {
        // ...
    }

}
```

Open Closed Principle

Software entities should be open for extension, but closed for modification.

- Minimize changes to existing code when adding new behavior
- Take advantage of object composition and polymorphism

```
public class Shape {
    // ...
}

public class Rectangle extends Shape {
    // ...
}

public class Circle extends Shape {
    // ...
}

public class GraphicEditor {

    public void drawShape(Shape s) {
        if (s instanceof Rectangle) {
            drawRectangle((Rectangle) s);
        } else if (s instanceof Circle) {
            drawCircle((Circle) s);
        }
    }

    public void drawRectangle(Rectangle rectangle) {
        // ...
    }

    public void drawCircle(Circle c) {
        // ...
    }

}
```

Open Closed Principle

Introduce abstraction.

- Tell, don’t ask
- Move responsibilities

```java
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```

Open Closed Principle

Introduce abstraction.

- Tell, don’t ask
- Move responsibilities

```
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```

Open Closed Principle

Introduce abstraction.

- Tell, don’t ask
- Move responsibilities

```java
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```

Open Closed Principle

Introduce abstraction.

- Tell, don't ask
- Move responsibilities

```java
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```

Open Closed Principle

Introduce abstraction.

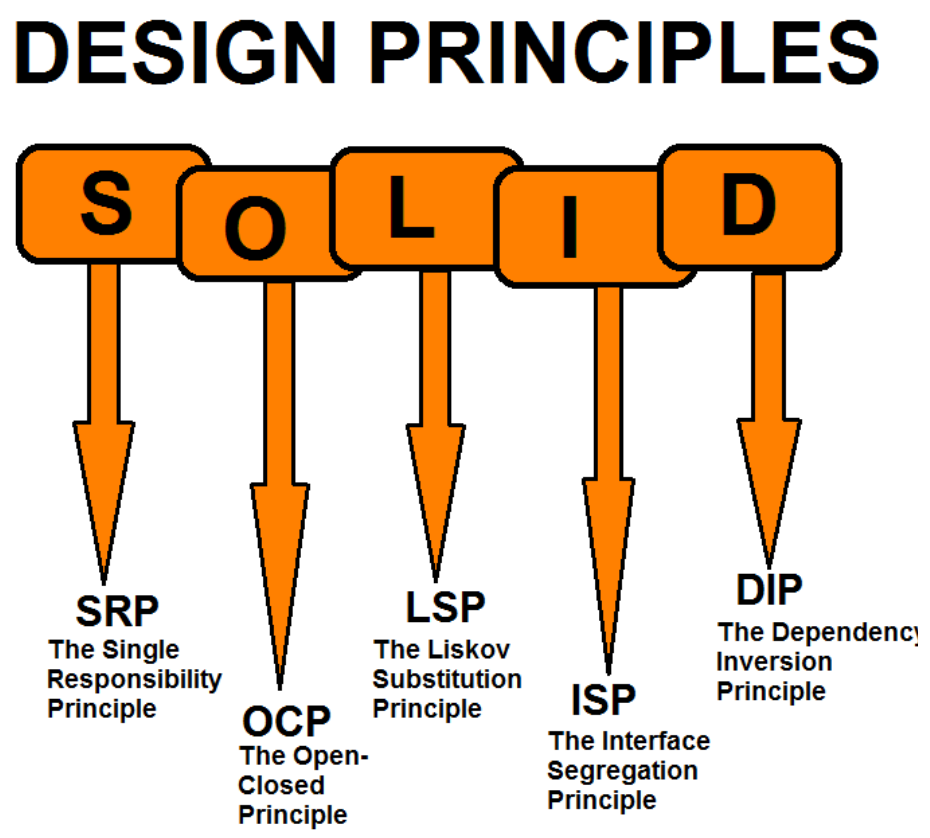
- Tell, don’t ask
- Move responsibilities

```java
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```

Exercises

Let's put S.O.L.I.D. principles into practice.

- Find principle violations in this code https://github.com/bebosudo/it.units.muli.poker
- Work on Fizz Buzz KATA with principles in mind (and don't forget about Simple Design and code smells)


DESIGN PRINCIPLES
S
O
L
I
D
SRP
The Single Responsibility Principle
OCP
The Open-Closed Principle
LSP
The Liskov Substitution Principle
ISP
The Interface Segregation Principle
DIP
The Dependency Inversion Principle

References

Refactoring
Martin Fowler

Extreme Programming: Explained
Kent Beck

Putting An Age-Old Battle To Rest
J. B. Rainsberger