



# Refactoring



Dario Campagna



# Why refactoring?



## Clean code

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



## No ugly 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?





---

# Readability

Small improvements in code readability can drastically improve code understandability

---

# Ways to improve readability

Atomic refactors

## Rename

Rename bad names, variables, arguments, instance variables, methods, classes.

Make abbreviations explicit.

## Extract

Constants from magic numbers and strings.

Conditionals.

Extract a class (or methods or variables...), creating a new abstraction.

## Inline

The inverse of extract – inline a method (or variable), deconstructing an abstraction.



# Ways to improve readability

Atomic refactors

## Move

Move a class (or methods or variables...) to some other place in the codebase.

## Safe delete

Delete code and its usages in the code base.

Delete unnecessary comments.

Delete dead code.

## Format

Format consistently and don't force the reader to waste time due to inconsistent formatting.



# Rename

The method name is accurate-but-vague.

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

Find

Display





# Rename

Now we have a precise name. Can we further improve readability?

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



# Extract

Two methods. Each with an intention-revealing name.

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

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



# Tennis Refactoring Kata

Clean-up the code to a point where someone can read it and understand it with ease.

<https://github.com/emilybache/Tennis-Refactoring-Kata>

- Work on the class “TennisGame1”
- The test suite provided is fairly comprehensive, and fast to run.
- You should not need to change the tests, only run them often as you refactor.





---

# Code Smells

Symptoms of a problem

---

# Code Smells

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

- Quick to spot
- Provide feedback on our decisions
- Don't always indicate a problem worth solving



# Categories of code smells

## Bloaters

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

## Object-orientation abusers

- Switch Statements
- Temporary Fields
- Refused Bequest
- Alternative Classes with Different Interfaces

## Couplers

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man

## Change preventers

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

## Dispensables

- Lazy Class
- Data Class
- Duplicated Code
- Dead Code
- Speculative Generality
- Comments

Have a look at [sourcemaking.com](http://sourcemaking.com)



# 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
- ...

```
1 package it.esteco.pos;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class Sale {
7
8     private Display display;
9     private final Map<String, String> pricesByBarcode;
10
11     public Sale(Display display, HashMap<String, String> pricesByBarcode) {
12         this.display = display;
13         this.pricesByBarcode = pricesByBarcode;
14     }
15
16     public void onBarcode(String barcode) {
17         if ("".equals(barcode)) {
18             display.setText("Scanning error: empty barcode!");
19         } else{
20             if (pricesByBarcode.containsKey(barcode)) {
21                 display.setText(pricesByBarcode.get(barcode));
22             } else {
23                 display.setText("Product not found for " +
24                     barcode);
25             }
26         }
27     }
28 }
```



# Feature Envy

A method accesses the data of another object more than its own data.

- Move method
- Extract method

```
1 public class Coordinate
2 {
3     public int X {get; set}
4     public int Y {get; set}
5 }
6
7 public class PositionUpdater
8 {
9     public Coordinate MoveUp(Coordinate coordinate)
10    {
11        return new Coordinate{X = coordinate.X, Y = coordinate.Y + 1};
12    }
13 }
```





# Message Chains

A message chain occurs when a client requests another object, that object requests yet another one, and so on.

- Hide delegate
- Extract method
- Move method

```
master.getModelisable()  
  .getDockablePanel()  
  .getCustomizer()  
  .getSaveItem()  
  .setEnabled(Boolean.FALSE.booleanValue());
```



```
master.allowSavingOfCustomizations();
```



# Smelly Tic Tac Toe

A TicTacToe implementation with quite a few code smells.

<https://github.com/dario-campagna/CodeSmells>

- Start by identifying the smells.
- Then slowly refactor the code.



# Exercise

Let's find some code smells.

[https://github.com/nicoleorzan/berlin\\_clock/blob/master/src/main/java/berlinclock](https://github.com/nicoleorzan/berlin_clock/blob/master/src/main/java/berlinclock)





---

# Coupling and Cohesion

Metrics that (roughly) describe how easy it will be to change the behavior of some code.

---



# Coupling

Measures the degree of interdependence between software components.

- Elements are coupled if a change in one forces a change in the other.
- We want to make changes in a component without impacting other components.
- We want coupling to be as low as possible, but not lower.



# Cohesion

Measures how strongly related and focused the responsibilities of a software module are.

- Indicates a component's functional strength and how much it focuses on a single point.
- Low cohesion results in behavior being scattered instead of existing in a single component.
- We want high cohesion.



LIFE Magazine (March 4, 1946)



# Cohesion, coupling and code smells

- Divergent Change
- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man
- Shotgun Surgery

## High coupling

Indicators of possible high coupling.

- Data Class
- Lazy Class
- Middle Man
- Primitive Obsession
- Shotgun Surgery

## Low cohesion

Indicators of possible low cohesion.





---

# S.O.L.I.D. Principles

Principle of class design

---

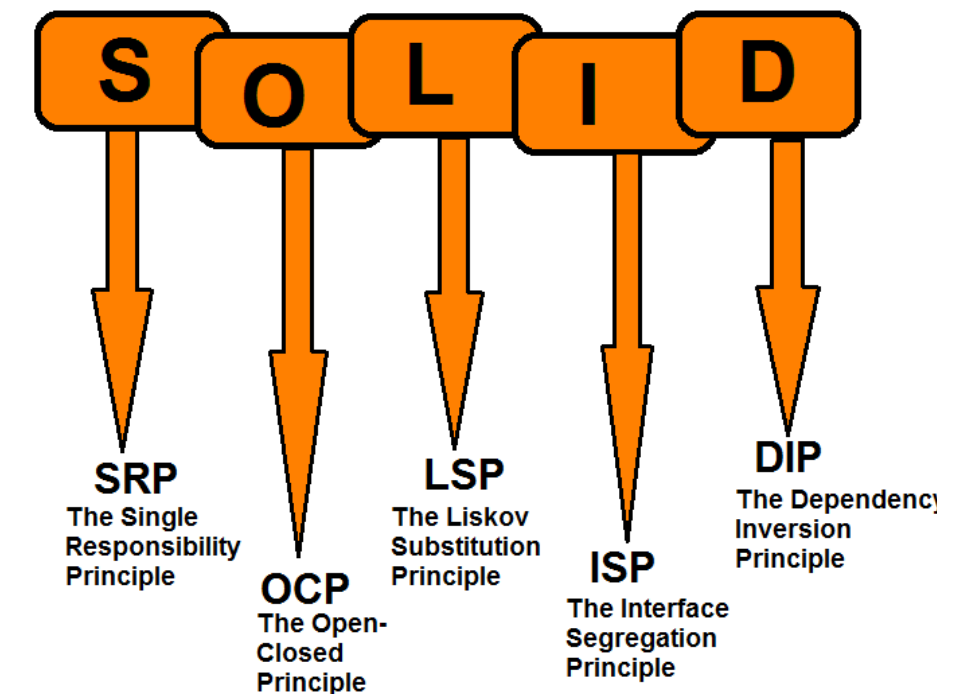


# S.O.L.I.D. Principles

Principle of class design that focus very tightly on dependency management.

- [Single Responsibility Principle](#)
- [Open-closed Principle](#)
- [Liskov Substitution Principle](#)
- [Interface Segregation Principle](#)
- [Dependency Inversion Principle](#)

## DESIGN PRINCIPLES



# Single Responsibility Principle

Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

- We want classes to be cohesive
- Only one reason to change
- 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
- Move 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  
    }  
}
```



# 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.

- Law of Demeter
- 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();  
    }  
}
```



# Dependency Inversion Principle

High level classes should not depend on low level classes.

- We want a flexible design
- We want to easily replace low level classes
- We want low coupling

```
public class Human {  
    public void work() {  
        // ...working  
    }  
}  
  
public class Manager {  
    private Human worker;  
  
    public void setWorker(Human worker) {  
        this.worker = worker;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}  
  
public class Robot {  
    public void work() {  
        // ...working longer  
    }  
}
```



# Dependency Inversion Principle

Introduce an abstraction that decouples the high-level and low-level classes from each other.

- High level classes depends on abstractions
- Low level classes are created based on abstractions

```
public interface Worker {  
    void work();  
}  
  
public class Human implements Worker {  
    public void work() {  
        // ...working  
    }  
}  
  
public class Robot implements Worker {  
    public void work() {  
        // ...working much more  
    }  
}  
  
public class Manager {  
    private Worker worker;  
  
    public void setWorker(Worker worker) {  
        this.worker = worker;  
    }  
    public void manage() {  
        worker.work();  
    }  
}
```

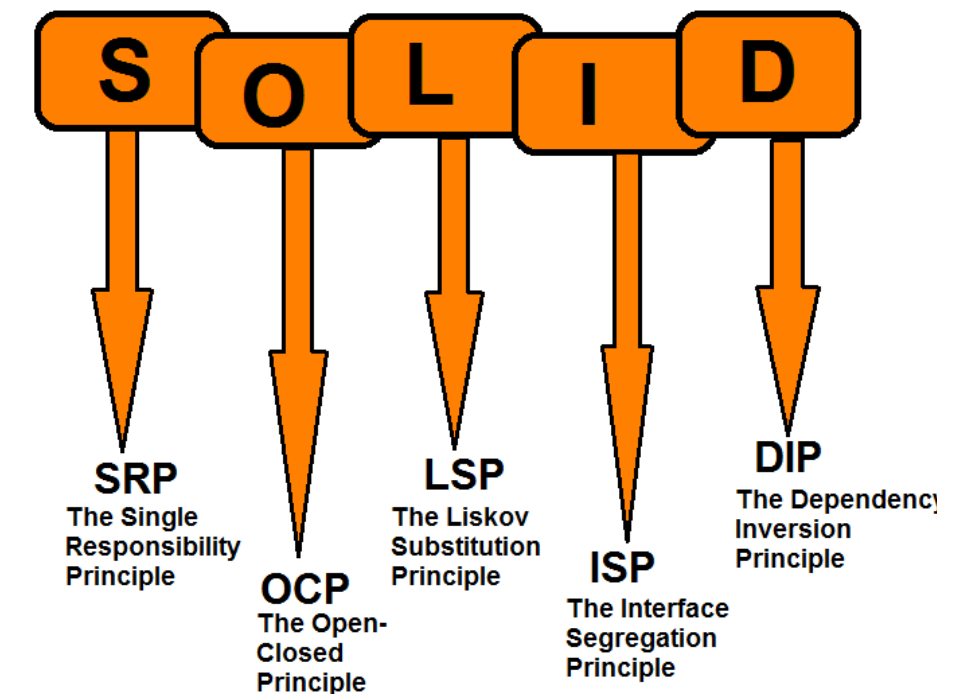


# Exercises

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

- Find principle violations in this project <https://github.com/bebosudo/it.units.muli.poker>
- Work on the [Cribbage Score Calculator assignment](#), use S.O.L.I.D. principles (and all the other concepts) when refactoring.

## DESIGN PRINCIPLES







---

# Simple Design

A goal/guide when refactoring

---

# Simple Design

According to Kent Beck, a design is “simple” if it follows this guidelines:

1. Passes the tests
2. Minimizes duplication
3. Reveals its intents
4. Has fewer classes/modules/packages...

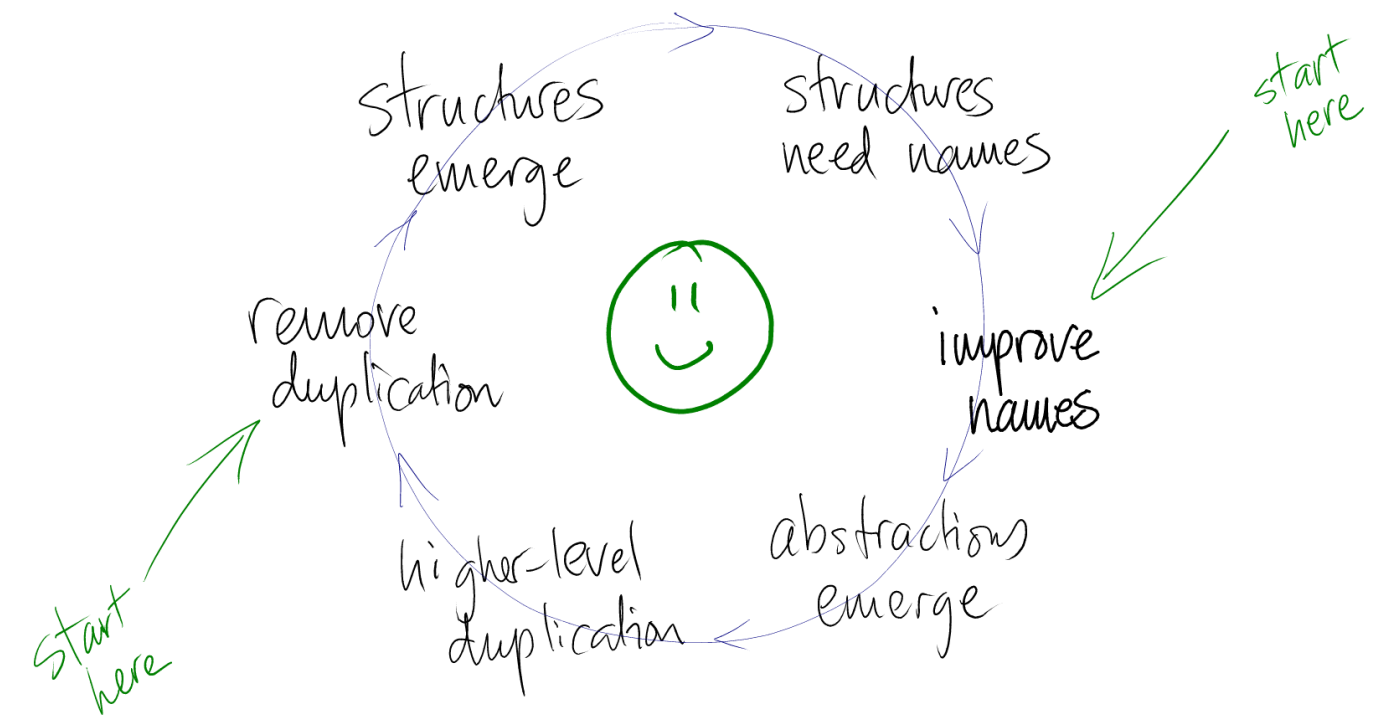


# The Simple Design Dynamo

Removing duplication and revealing intent/  
increasing clarity quickly form a rapid, tight  
feedback cycle.

[Putting An Age-Old Battle To Rest, J.B. Rainsberger](#)

- When we remove duplication, we create buckets.
- When we improve names, we create more cohesive, more easily-abstracted buckets.



©jbrains 2013



# References



## **Agile Technical Practices Distilled**

Pedro Moreira Santos, Marco Consolaro,  
Alessandro Di Gioia

## **Refactoring**

Martin Fowler



Go refactor!

[esteco.com](http://esteco.com)

