



Refactoring, S.O.L.I.D. Principles and Simple Design



Dario Campagna

Head of Research and Development



S.O.L.I.D. Principles

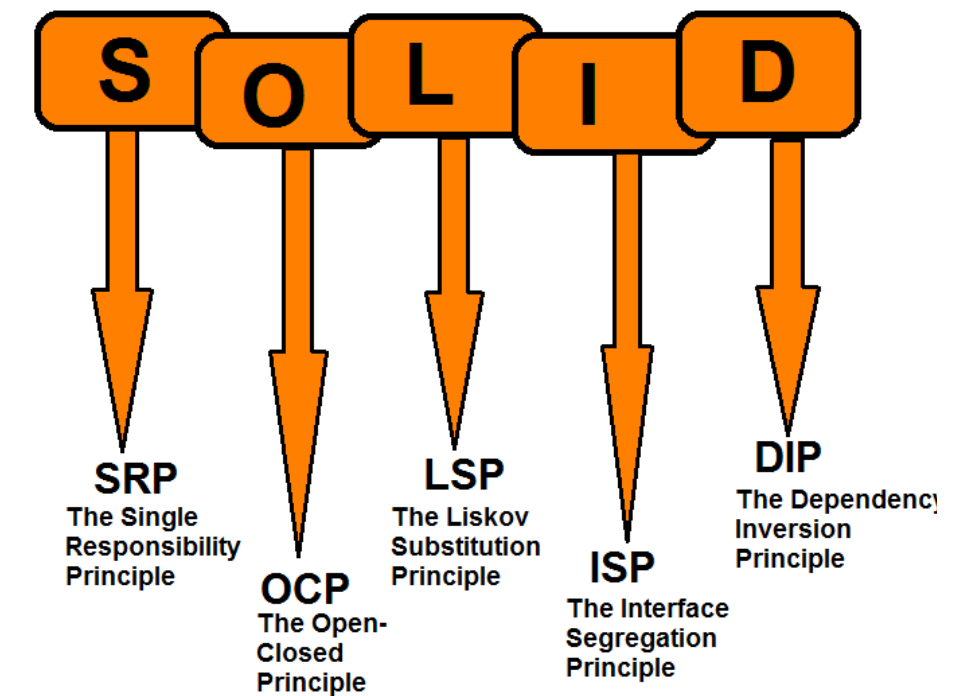
Principle of class design

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();  
    }  
}
```

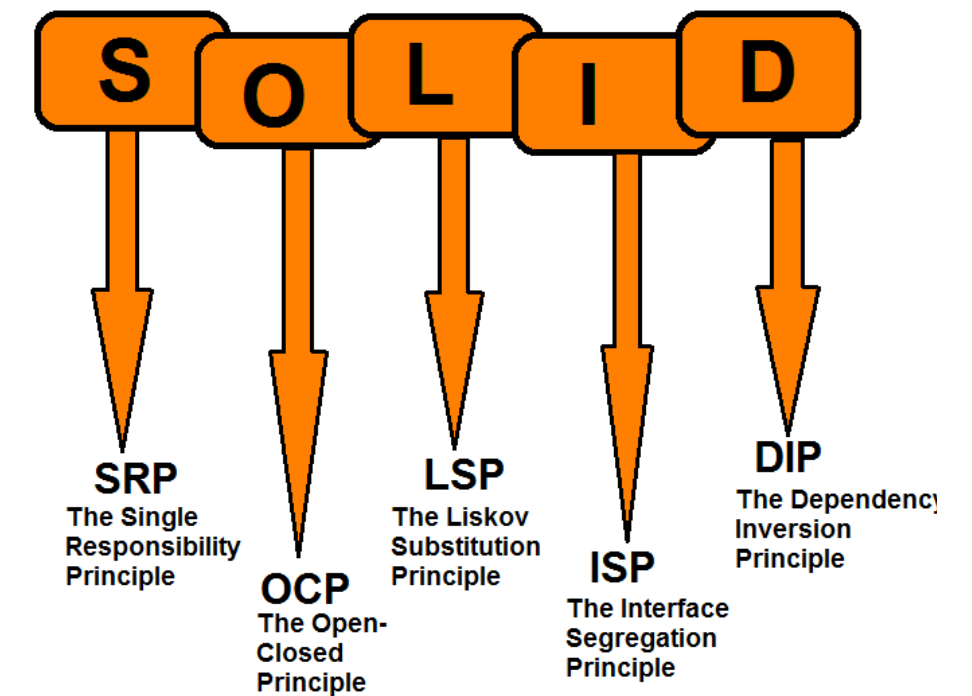


Exercises

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 [Tennis Refactoring Kata](#), 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...

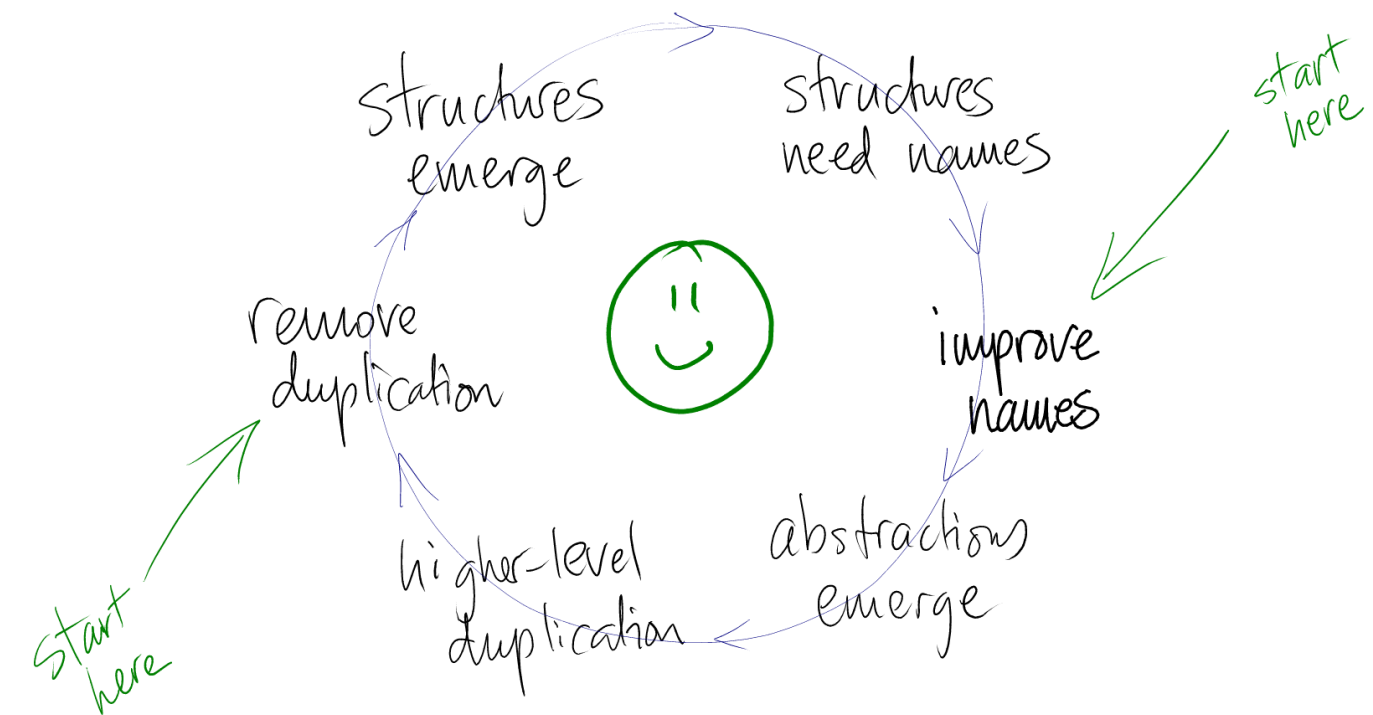


The Simple Design Dynamo

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

