S.O.L.I.D.

Jacek Wasilewski

S.O.L.I.D.

- S.O.L.I.D. stands for
  - S – Single-Responsibility Principle
  - O – Open-Closed Principle
  - L – Liskov Substitution Principle
  - I – Interface Segregation Principle
  - D – Dependency Inversion Principle
- Proposed by Robert C. Martin aka Uncle Bob

Single-Responsibility

- A class should have one and only one reason to change, meaning that a class should have only one job.
- Problem: consider a module that compiles and prints a report
- Code

```
class Report {
  def calculate = 42 // really complex computations
  def output = println("Report: " + calculate)
}
```

Single-Responsibility

- Report with different calculations

```
class Report2 {
  def calculate = 24 // really complex computations
  def output = println("Report: " + calculate)
}
```

- Report with different output

```
class Report3 {
  def calculate = 42 // really complex computations
  def output = "<b>Report:</b> " + calculate
}
```

Single-Responsibility

- These changed for very different causes: substantive, and cosmetic
- According to SRP, these two aspects are really two separate responsibilities, and should therefore be in separate classes
- Code

```
class Report {
  def calculate = 42
}
class ReportOutputter(report: Report) {
  def output = println("Report: " + report.calculate)
}
```

Open-Closed

- Objects or entities should be open for extension, but closed for modification
- Simply saying, class should be easily extendable without modifying the class itself

Open-Closed

```
abstract class Shape
case class Rectangle() extends Shape
case class Circle() extends Shape
class GraphicEditor {
  def drawShape(s: Shape) = s match {
    case r: Rectangle => drawRectangle(r)
    case c: Circle => drawCircle(c)
  }
  def drawRectangle(r: Rectangle) = ???
  def drawCircle(c: Circle) = ???
}
```

Open-Closed

```
abstract class Shape {
  def draw
}
case class Rectangle() extends Shape {
  orverride def draw = ???
}
case class Circle() extends Shape {
  override def draw = ???
}
class GraphicEditor {
  def drawShape(s: Shape) = s.draw
}
```

Liskov Substitution

- Let q(x) be a property provable about objects of x of type T
- Then q(y) should be provable for objects y of type S where S is a subtype of T
- Simply saying, every subclass should be substitutable for their super-class

Liskov Substitution

- Square is a Rectangle

  ```
  class Rectangle(height: Int, weight: Int)
  class Square(side: Int) extends Rectangle(side, side)
  ```

- Square can be used anywhere Rectangle is expected
- If anomalies arise, you might have wrong abstraction

Liskov Substitution

```
class Rectangle(height: Int, weight: Int) {
  var h = height
  var w = weight
  def setH(hh: Int) = { h = hh }
  def setW(ww: Int) = { w = ww }
}
class Square(side: Int) extends Rectangle(side, side) {
  def setS(s: Int) = { h = s; w = s }
}
```

Liskov Substitution

```
val r: Rectangle = new Rectangle(1, 2)
r.setH(2)
r.setW(4)

val r2: Rectangle = new Square(4)
r2.setH(2)
r2.setW(4)
```

Interface Segregation

- A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use
- Let's say we have a 3D shape, cube, and interface for shapes

```
abstract class Shape {
  def area
  def volume
}
```

Interface Segregation

- Then
```
class Cube(x: Int) extends Shape {
  def area = 6 * x * x
  def volume = x * x * x
}
```
- Let's say we want to have a square:
```
class Square(x: Int) extends Shape {
  def area = x * x
  def volume = ???
}
```
- Need to implement everything even if not needed

Interface Segregation

```
trait Shape {
  def area
}
trait SolidShape {
  def volume
}
class Cube(x: Int) extends Shape with SolidShape {
  def area = 6 * x * x
  def volume = x * x * x
}
class Square(x: Int) extends Shape {
  def area = x * x
}
```

Dependency Inversion

- Entities must depend on abstractions not on concretions
- It states that the high level module must not depend on the low level module, but they should depend on abstractions
- Communication through interfaces

Dependency Inversion

```
class Worker {
  def work = ??? // working
}
class Manager {
  def manage(w: Worker) {
    w.work()
  }
}
class HardWorker {
  def work = ??? // working hard
}
```

Dependency Inversion

```
abstract class Worker {
  def work
}
class NormalWorker extends Worker {
  def work = ??? // working
}
class HardWorker extends Worker {
  def work = ??? // working hard
}
class Manager {
  def manage(w: Worker) {
    w.work
  }
}
```