SOLID

OBJECT ORIENTED DESIGN PRINCIPLES

Enes AYKURT
METU Computer Engineering

SOLID

- Five Principles Of Class Design
- First introduced by Michael Feathers for the "First five principles“
- Identified by Robert C. Martin in early 2000s

SOLID (Five Principles Of Class Design);

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

- A class should have only one responsibility
- The more a class does, the more likely it will change
- "*There should never be more than one reason for a class to change.*" — Robert Martin

Single Responsibility Principle (SRP)

Two resposibilities:

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

Connection Management + Data Communication

Single Responsibility Principle (SRP)

Separate into two interfaces:

```
interface DataChannel {
 public void send(char c);
 public char recv();
}


interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

Open Closed Principle (OCP)

- *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." — Robert Martin paraphrasing Bertrand Meyer*

- **Open for extension;**

Behavior of a class can be changed by using inheritance, composition (or aggregation), interfaces, abstractions etc.

- **Closed for modification;**

Does **not** change the class itself.

Open Closed Principle (OCP)

```
// Open-Close Principle - Bad example
 class GraphicEditor {
         public void drawShape(Shape s) {
                  if (s.m_type==1)
                           drawRectangle(s);
                  else if (s.m_type==2)
                           drawCircle(s);
                  }

         }
         public void drawCircle(Circle r) {....}
         public void drawRectangle(Rectangle r) {....}

 }

 class Shape {
         int m_type;
 }

 class Rectangle extends Shape {
         Rectangle() {
                  super.m_type=1;
         }
 }

 class Circle extends Shape {
         Circle() {
                  super.m_type=2;
         }
 }
```

Open Closed Principle (OCP)

- Impossible to add a new `Shape` without modifying `GraphEditor` (Tight coupling between `GraphEditor` and `Shape`)

- Each change on a class can introduce bugs and requires re-testing of this class

- Difficult to test a specific `Shape` without involving `GraphEditor`

Open Closed Principle (OCP)

```
// Open-Close Principle - Good example
class GraphicEditor {
	public void drawShape(Shape s) {
		s.draw();
	}
}

class Shape {
	abstract void draw();
}

class Rectangle extends Shape {
	public void draw() {
	// draw the rectangle
	}
}
```

Liskov Substitution Principle (LSP)

- *"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." — Robert Martin*

- Calling code should not know that one class is different from its substitute

- Non-substitutable code breaks polymorphism

Liskov Substitution Principle (LSP)

```
//Violation of Liskov's Substitution
 Principle
class Rectangle
{
        int m_width;
        int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int h){
                m_height = ht;
        }

        public int getWidth(){
                return m_width;
        }

        public int getHeight(){
                return m_height;
        }

        public int getArea(){
        return m_width * m_height;
        }
}
```

```
class Square extends Rectangle
 {
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }
 }
```

Liskov Substitution Principle (LSP)

```
class LspTest
{
    private static Rectangle getNewRectangle()
    {
        // it can be an object returned by some factory ...
        return new Square();
    }

public static void main (String args[])
    {
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);

        // user knows that r it's a rectangle. It assumes

        // that he's able to set the width and height as

        // for the base class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
```

Interface Segregation Principle (ISP)

- *"Clients should not be forced to depend upon interfaces that they do not use." — Robert Martin*

- Prefer small, cohesive interfaces to "fat" interfaces (for the implementer and caller)

Interface Segregation Principle (ISP)

```
//bad example
 //(polluted interface)
 interface Worker {
  void work();
  void eat();
 }
ManWorker implements Worker
{
  void work() {…};
  void eat() {
       30 min break;
  };
}
```

```
RobotWorker implements Worker
{
  void work() {…}
  void eat() {
  //Not Appliciable for
  //a RobotWorker
  }
}
```

Interface Segregation Principle (ISP)

- Solution; split into two interfaces

```
interface Workable {
	public void work();
}

interface Feedable{
	public void eat();
}
```

Dependency Inversion Principle (DIP)

- *"A. High level modules should not depend upon low level modules. Both should depend upon abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions." — Robert Martin*

- In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built

Dependency Inversion Principle (DIP)

```
//DIP - bad example
public class EmployeeService {
        //concrete class, not abstract. Can access a SQL DB for instance
        private EmployeeFinder emFinder
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```

Dependency Inversion Principle (DIP)

```
//DIP - fixed
public class EmployeeService {
        //depends on an abstraction, not an implementation
        private IEmployeeFinder emFinder;

        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```

Now its possible to change the finder to be a XmlEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder, MockEmployeeFinder….

Q&A

References:

http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

http://social.technet.microsoft.com/wiki/contents/articles/18033.software-design-principles.aspx

"SOLID - OO DESIGN PRINCIPLES" Presentation by Andreas Enbohm

"SOLID Object Oriented Design" Presentation by Craig Berntson

"Refactoring Code to a SOLID Foundation" Presentation by Adnan Masood

http://en.wikipedia.org/wiki/Open/closed_principle

http://www.oodesign.com/interface-segregation-principle.html

http://en.wikipedia.org/wiki/Dependency_inversion_principle