The SOLID OOP Principles

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

April 28, 2022

Outline

Outline

Project Meeting

Before next project iteration, each group should schedule a meeting with me in this or the next week – more scheduling details will be on Blackboard.

Agenda and Objectives

- Plan for next iteration
- Discuss group and individual progress
- Identify gaps and improvements
- Any issues you may have regarding the class

Outline

SOLID Classes

Motivation: in order to minimize cost of change, we should to design classes that are SOLID

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Interface Segregation Principle
- Demeter principle

SOLID Classes

Motivation: in order to minimize cost of change, we should to design methods that are SOFA, i.e.,

- the methods are Short,
- do One thing,
- have Few arguments, and
- have single level of Abstraction

SOLID concerns itself with designing classes, assuming the methods are already SOFA

Outline

Single Responsibility Principle

A class should have one and only one reason to change

- Each responsibility is a possible axis of change
- Changes to one axis shouldn't affect others

What is class's responsibility, in a sensor or two?

- Part of the craft of OO design is defining responsibilities and then sticking to them

Let's consider examples:

- An instance of the User class is a moviegoer, and an authentication principal, and a social network member, ... Bad!

Detecting Violations

We can examine some metrics to detect possible violations of the Single Responsibility Principle

- Lines of Code
  - Usually, really big class files are a tip-off – lines of Code is a coarse metric to detect this
- Lack of Cohesion of Methods (LCOM)

Lack of Cohesion of Methods (LCOM)

A simple method to estimate LCOM,

$$LCOM = 1 - \frac{\sum(m_i)}{MV} \tag{1}$$

where

- $LCOM \in [0, 1]$
- M $= \#$ of methods of the class
- V $= \#$ of instance or class variables of the class
- $m_i = \#$ of class methods that access the i'th class variable

Observations:

- A class is utterly cohesive if all its methods use all its instance fields, because $\sum(m_i) = MV$ and then LCOM $= 0$
- High LCOM suggests possible violation of the Single Responsibility Principle

Discussion Question – which one is true?

Which one is true with regard to the Single Responsibility Principle (SRP)?

- If a class respects SRP, its methods probably respect SOFA
- If a class's methods respect SOFA, the class probably respects SRP

Outline

The Open-Closed Principle

Classes should be *open for extension*, but *closed for source modification*.

Discussion Question

Is the following class *open for extension*, but *closed for source modification*?

```
1 class Report {
2   void outputReport(String outputFormat) {
3     switch (outputFormat) {
4       case "html":
5         HtmlFormatter.output(this);
6         break;
7       case "pdf":
8         PdfFormatter.output(this);
9     }
10 }
```

Discussion Question

Is the following class *open for extension*, but *closed for source modification*?

```
1 class Report {
2   void outputReport(String outputFormat) {
3     switch (outputFormat) {
4       case "html":
5         HtmlFormatter.output(this);
6         break;
7       case "pdf":
8         PdfFormatter.output(this);
9     }
10 }
```

Not the best, because can't extend format (add new formatter types) without changing `Report` class or know what its implementation details in the `outputReport` method.

How to improve the design?

Discussion Question

Using the Strategy design pattern, we redesign the class as follows,

```
 1 class Report {
 2   Report(Formatter f) {
 3     formatter = f;
 4   }
 5
 6   void outputReport() {
 7     formatter.output(this);
 8   }
 9 }
10
11 interface Formatter {
12   void output(Report report);
13 }
14
15 class PdfFormatter implements Formatter {...}
16 class HtmlFormatter implements Formatter {...}
17 class MsdocFormatter implements Formatter {...}
```

Any alternative way to redesign the class?

Practical Considerations

- Can't close against all types of changes, so have to choose (make a design decision), and we might make a wrong decision
- Agile methodology can help expose important types of changes early
  - Scenario-driven design with prioritized features
  - Short iterations
  - Test-first development

  Then we can try to apply the principle for those types of changes we identify in each iteration

Outline

Liskov Substitution Principle

Attributed to Barbara Liskov

"A method that works on an instance of type T, should also work on any subtype of T"

Note that in dynamically typed languages often type/subtype != class/subclass

Discussion Question – does this follow the principle?

Source of the example: http://javacodegeeks.com/)

```
 1 class Bird {
 2   public void fly(){}
 3   public void eat(){}
 4 }
 5 class Crow extends Bird {}
 6 class Ostrich extends Bird{
 7   fly(){ throw new UnsupportedOperationException(); }
 8 }
 9 public BirdTest{
10   public static void main(String[] args){
11     List<Bird> birdList = new ArrayList<Bird>();
12     birdList.add(new Bird());
13     birdList.add(new Crow());
14     birdList.add(new Ostrich());
15     letTheBirdsFly ( birdList );
16   }
17   static void letTheBirdsFly ( List<Bird> birdList ){
18     for ( Bird b : birdList ) { b.fly(); }
19   }
20 }
```

Discussion Question – does this follow the principle?

How do we redesign it? Let's consider additional design principles and patterns ...

- Dependency Inversion Principle

Dependency Inversion Principle

Problem: A depends on B, but B's interface & implementation can change, even if functionality is stable

Solution: insert an abstract interface that A & B depend on

- An example of the Observer, Adapter, or sometimes Facade design pattern
- Dependence Inversion: now B (and A) depend on interface, vs. A depending on B – the dependencies are inversed

Outline

Interface Separation Principle

Clients should not be forced to depend on interfaces they do not need.

Discussed before

Discussion Question – How do we redesign this?

Source of the example: http://javacodegeeks.com/)

```
 1 class Bird {
 2   public void fly(){}
 3   public void eat(){}
 4 }
 5 class Crow extends Bird {}
 6 class Ostrich extends Bird{
 7   fly(){ throw new UnsupportedOperationException(); }
 8 }
 9 public BirdTest{
10   public static void main(String[] args){
11     List<Bird> birdList = new ArrayList<Bird>();
12     birdList.add(new Bird());
13     birdList.add(new Crow());
14     birdList.add(new Ostrich());
15     letTheBirdsFly ( birdList );
16   }
17   static void letTheBirdsFly ( List<Bird> birdList ){
18     for ( Bird b : birdList ) { b.fly(); }
19   }
20 }
```

Outline

The Demeter Principle

A class should only know about the methods of other classes, not their internals

The Demeter Principle

What should we consider for this principle?

- e.g., avoid getter methods
- In practice: only talk to our friends ... not strangers, which means,
  - We can call methods on ourselves, use our own instance variables and parameters passed to the method
  - But not on the results returned by them

Discussion Question

Does this violate Demeter?

```
1   Options options = context.getOptions();
2   File scratchDir = opts.getScratchDir();
3   final string outputDir = scratchDir.getAbsolutePath();
```

or equivalently,

```
1   final string outputDir = context.getOptions()
2                                   .getScratchDir()
3                                   .getAbsolutePath();
```

Discussion Question

Does this violate Demeter?

- Yes, if we consider Options and File as objects

```
1   Options options = context.getOptions();
2   File scratchDir = opts.getScratchDir();
3   final string outputDir = scratchDir.getAbsolutePath();
```

or equivalently,

```
1   final string outputDir = context.getOptions()
2                                   .getScratchDir()
3                                   .getAbsolutePath();
```

How do we fix this?

Discussion Question

Examining what we really use the otuputDir for, we revise the context manager so that,

```
1 BufferedOutputWriter outputWriter = ctxt.getOutputWriter()
```

Summary and Questions

SOLID

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Interface Segregation Principle
- Demeter principle

Dependency Inversion Principle

Let's do an exercise ...

Outline

"Engineering Software as a Service" by Armando Fox and David Patterson (2nd Edition)
"Introduction to Software Design with Java" by Martin P. Robillard
"Essentials of Software Engineering" by Frank Tsui, Orlando Karam, and Barbara Bernal(4th Edition)