Design Characteristics and Metrics: Part I

Hui Chen [a]

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

May 9, 2023

Outline

Outline

SOLID and SOFA Design

Motivation: in order to minimize cost of change, we want classes to be SOLID and methods to be SOFA.

Outline

SOFA Methods

Motivation: in order to minimize cost of change, we want to design methods to be SOFA, i.e.,

- the methods are **S**hort,
- do **O**ne thing,
- have **F**ew arguments, and
- have single level of **A**bstraction

Single Level of Abstraction (SLAP)

Code within a method should be at the same level of abstraction, i.e, parts of the method should be on the level of "details".

To understand this, let's take a look at two examples.

Counter Example 1

```
 1   void dispGpa() {
 2     // read to this.gradeList
 3     readCourseData();
 4     int totalPoints = 0;
 5     for (Grade grade: gradeList) {
 6       if (grade.getGrade().equals("A")) {
 7         totalPoints += 4 * grade.getCredits();
 8       } else {
 9         // ... more ...
10       }
11     }
12     double gpa = //... more ...
13     displayGpa(gpa);
14   }
15
```

Are all parts on the same level of abstraction?

Counter Example 1: Refactoring to be SOFA

```
1   void dispGpa() {
2     // read to this.gradeList
3     readCourseData();
4     double gpa = computeGpa();
5     displayGpa(gpa);
6   }
7
8   double computeGpa() {
9     // ...
10  }
11
```

Counter Example 2

The following example is due to Neal Ford[1].

[1] Neal Ford. *The productive programmer*. " O'Reilly Media, Inc.", 2008.

Counter Example 2: Source Code

```
1  public void addOrder(ShoppingCart cart, String userName, Order
       order) throws SQLException {
2    Connection c = null; PreparedStatement ps = null;
3    Statement s = null; ResultSet rs = null;
4    boolean transactionState = false;
5    try {
6      c = dbPool.getConnection();
7      s = c.createStatement();
8      transactionState = c.getAutoCommit();
9      int userKey = getUserKey(userName, c, ps, rs);
10     c.setAutoCommit(false);
11     addSingleOrder(order, c, ps, userKey);
12     int orderKey = getOrderKey(s, rs);
13     addLineItems(cart, c, orderKey);
14     c.commit();
15     order.setOrderKeyFrom(orderKey);
16   } catch (SQLException sqlx) {
17     s = c.createStatement(); c.rollback(); throw sqlx;
18     // to be continued
```

Counter Example 2: Source Code: Continued

```
19    } finally {
20      try {
21        c.setAutoCommit(transactionState);
22        dbPool.release(c);
23        if (s != null) s.close();
24        if (ps != null) ps.close();
25        if (rs != null) rs.close();
26      } catch (SQLException ignored) { }
27    }
28  }
```

Are these on the same abstraction level?

Counter Example 2: Critique

The `addOrder` method contains the following parts:

- detailed steps to set up database infrastructure
- higher-level business domain methods like `addSingleOrder`
- ...

The code is hard read because it jumps between abstraction levels almost randomly, based on what steps need to occur next.

Counter Example 2: Refactoring to be SOFA

Neal Ford refactored it into the following[2]:

[2] Neal Ford. *The productive programmer*. " O'Reilly Media, Inc.", 2008.

Counter Example 2: Refactoring to be SOFA

Neal Ford refactored it into the following[3]:

[3] Neal Ford. *The productive programmer*. " O'Reilly Media, Inc.", 2008.

Counter Example 2: Refactored Code: 1/4

```
1   public void addOrderFrom(ShoppingCart cart, String userName,
       Order order) throws SQLException {
2     Map db = setupDataInfrastructure();
3     try {
4       int userKey = userKeyBasedOn(userName, db);
5       add(order, userKey, db);
6       addLineItemsFrom(cart,
7       order.getOrderKey(), db);
8       completeTransaction(db);
9     } catch (SQLException sqlx) {
10      rollbackTransactionFor(db);
11      throw sqlx;
12    } finally {
13      cleanUp(db);
14    }
15  }
```

Counter Example 2: Refactored Code: 2/4

```
1   private Map setupDataInfrastructure() throws SQLException {
2     HashMap db = new HashMap();
3     Connection c = dbPool.getConnection();
4     db.put("connection", c);
5     db.put("transaction state",
6     Boolean.valueOf(setupTransactionStateFor(c)));
7     return db;
8   }
```

Counter Example 2: Refactored Code: 3/4

```
1   private void cleanUp(Map db) throws SQLException {
2     Connection connection = (Connection) db.get("connection");

3     boolean transactionState = ((Boolean)
4     db.get("transation state")).booleanValue();
5     Statement s = (Statement) db.get("statement");
6     PreparedStatement ps = (PreparedStatement)
7     db.get("prepared statement");
8     ResultSet rs = (ResultSet) db.get("result set");
9     connection.setAutoCommit(transactionState);
10    dbPool.release(connection);
11    if (s != null) s.close();
12    if (ps != null) ps.close();
13    if (rs != null) rs.close();
14  }
```

Counter Example 2: Refactored Code: 4/4

```
1   private void rollbackTransactionFor(Map dbInfrastructure)
      throws SQLException {
2       ((Connection) dbInfrastructure.get("connection")).
      rollback();
3   }
4
5   private void completeTransaction(Map dbInfrastructure)
      throws SQLException {
6     ((Connection) dbInfrastructure.get("connection")).commit()
      ;
7   }
8
9   private boolean setupTransactionStateFor(Connection c)
      throws SQLException {
10    boolean transactionState = c.getAutoCommit();
11    c.setAutoCommit(false);
12    return transactionState;
13  }
```

Outline

SOLID Design

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

(Note: some of which have some variations)

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

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 sense 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 movie-goer, 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) (to be discussed)

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

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

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 (Interface Separation Principle)

Additionally, we may also consider (a brief info, detailed exploration on your own)

- Adapter Design Pattern
- Facade Design Pattern
- ...

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

Adapter Design Pattern

Problem: client wants to use a "service"

- The service generally supports desired operations,
- but the API's don't match what client expects
- and/or client must interoperate transparently with multiple slightly-different services

Explore this on your own.

Facade Design Pattern

If the hidden/dependent functionality is more than just one class, e.g.,

- for the mail list application: initialization, list management, start/stop campaign ...

Then the adapted becomes a facade unifies distinct underlying API's into a single, simplified API

Explore this on your own.

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 }
```

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 `outputDir` 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

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)