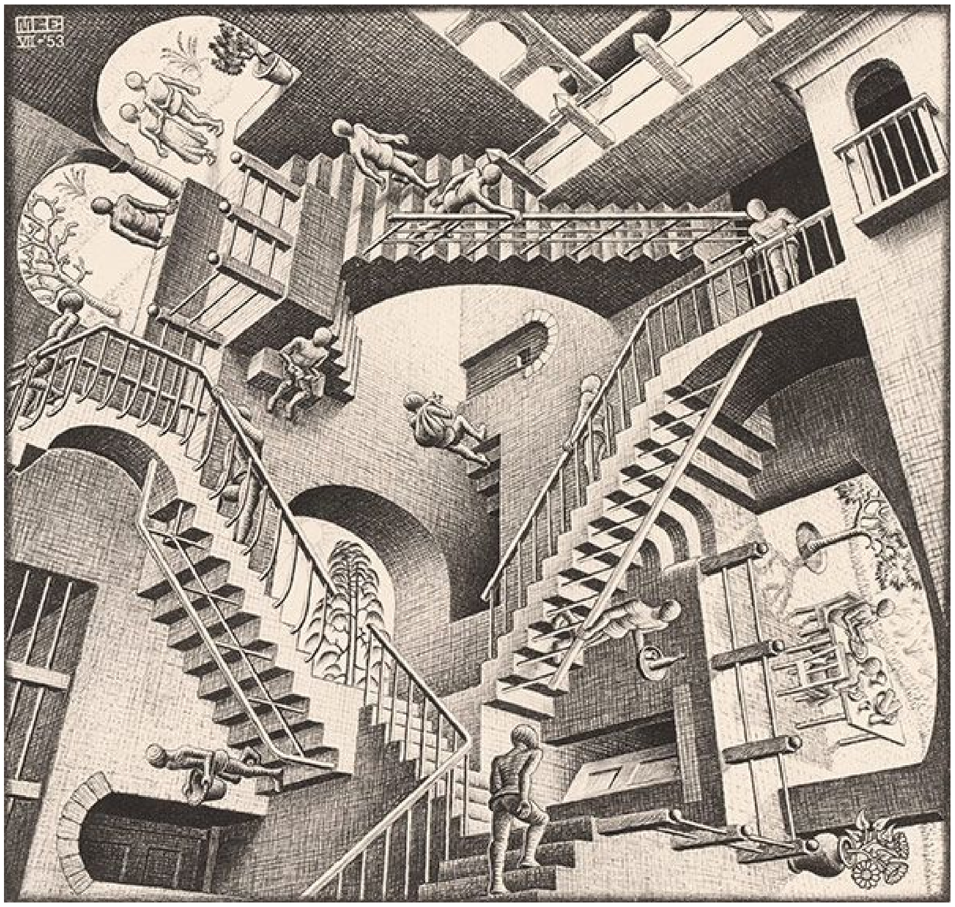
Lecture Notes
October 15, 2020

Working in a team: software *design*

Here is one approach to design:



The design manifest in this lithograph is reminiscent of spaghetti code: a great idea for art, a really bad idea for software engineering.

A *design principle* is a general technique to avoid bad design (or, phrased positively, create good designs)

Bad design is:

- Rigid - Hard to change the code because every change affects too many other parts of the system.
- Fragile - When you make a change to some part of the code, unexpected parts of the system break.
- Immobile - Hard to reuse some of the code in another application because it cannot be disentangled from the current application.

You Aren’t Gonna Need It (YAGNI): do the simplest design that could possibly work for the functionality needed NOW

“SOLID” Software Engineering Design Principles =

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

Some of these apply to any language, some assume OO

Single Responsibility Principle (SRP): Every entity or class should have a single responsibility, and all its tasks or services should be focused on that responsibility

Seemingly contradicts the multiple “responsibilities” of CRC cards, but in CRC ideally refers to tasks or services that are all part of the same focused single responsibility. Might have been better to call it the Single Purpose Principle, or Single Job Principle.

When the requirements of the project change, this will change the purpose of one or more existing classes or require one or more new classes.

Problem: If a class has multiple purposes, then it is more likely to be affected by a requirements change. It knows too much.

Solution: Break up the functionality into multiple classes, where each individual class has only one purpose. Leads to more smaller classes

Greater cohesion, lower coupling, fewer test cases

```
public class TextManipulator {
    private String text;

    public TextManipulator(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public void appendText(String newText) {
        text = text.concat(newText);
    }

    public String findWordAndReplace(String word, String
replacementWord) {
        if (text.contains(word)) {
            text = text.replace(word, replacementWord);
        }
        return text;
    }

    public String findWordAndDelete(String word) {
        if (text.contains(word)) {
            text = text.replace(word, "");
        }
        return text;
    }
```

What is wrong with this?

```
public class TextManipulator {
    private String text;

    ...

    public void printText() {
    System.out.println(textManipulator.getText());
    }
}
```

Is this better?

```
public class TextPrinter {
   TextManipulator textManipulator;

   public TextPrinter(TextManipulator textManipulator) {
      this.textManipulator = textManipulator;
   }

   public void printText() {
      System.out.println(textManipulator.getText());
   }

   public void printOutEachWordOfText() {

System.out.println(Arrays.toString(textManipulator.getText().split("
")));
   }

   public void printRangeOfCharacters(int startingIndex, int
endIndex) {

System.out.println(textManipulator.getText().substring(startingInd
ex, endIndex));
   }
}
```

Open-Closed Principle (OCP): Code units should be "open for extension and closed for modification". This allows change without modifying any existing code (except for fixing bugs in the existing code)

Once you have code that works, and is being used, don't make changes to it that might introduce new bugs

But since requirements change, we need to be able to change existing program behavior

- Change the behavior of an existing class (in an OO language) by deriving a new subclass

- Change the behavior of an existing function (or procedure, subroutine, method, etc.) by adding a wrapper around that function

- Change the behavior of a library by adding new API functions or a new implementation of the same interface

```
public interface CalculatorOperation {}

public class Addition implements CalculatorOperation {
    private double left;
    private double right;
    private double result = 0.0;

    public Addition(double left, double right) {
        this.left = left;
        this.right = right;
    }

}

public class Subtraction implements CalculatorOperation {
    private double left;
    private double right;
    private double result = 0.0;

    public Subtraction(double left, double right) {
        this.left = left;
        this.right = right;
    }

}
```

What is wrong with this?

```
public class Calculator {

    public void calculate(CalculatorOperation operation) {
        if (operation == null) {
            throw new InvalidParameterException("Can not perform
operation");
        }

        if (operation instanceof Addition) {
            Addition addition = (Addition) operation;
            addition.setResult(addition.getLeft() + addition.getRight());
        } else if (operation instanceof Subtraction) {
            Subtraction subtraction = (Subtraction) operation;
            subtraction.setResult(subtraction.getLeft() -
subtraction.getRight());
        }
    }
}
```

How would we add multiplication and division?

Is this better?

```
public interface CalculatorOperation {
    void perform();
}

public class Addition implements CalculatorOperation {
    private double left;
    private double right;
    private double result;

    @Override
    public void perform() {
        result = left + right;
    }
}

public class Calculator {

    public void calculate(CalculatorOperation operation) {
        if (operation == null) {
            throw new InvalidParameterException("Cannot perform
operation");
        }
        operation.perform();
    }
}
```

```
public class Division implements CalculatorOperation {
    private double left;
    private double right;
    private double result;

    @Override
    public void perform() {
        if (right != 0) {
            result = left / right;
        }
    }
}
```

Liskov Substitution Principle (LSP): Extends OCP to require that subtypes must be substitutable for their base types (polymorphism)

if class *A* is a subtype of class *B*, then we should be able to replace *B* with *A* without disrupting the behavior of our program

Do inherited methods make sense for instances of the derived class? Do overridden methods take the same parameter types and return the same types as the base class?

If not, then we are not actually inheriting - instead should delegate any common functionality, and if there is no common functionality then there's no relationship at all so why pretend?

Code already using references to the base classes should be able to use instances of the derived classes without being aware of the switch

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}

public class MotorCar implements Car {

    private Engine engine;

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

What is wrong with this?

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Is this better?

```
public interface Car {
   void accelerate();
}

public class MotorCar implements Car {
   private Engine engine;

 public void turnOnEngine() {
      //turn on the engine!
      engine.on();
   }
   public void accelerate() {
      //move forward!
      engine.powerOn(1000);
   }
}

public class ElectricCar implements Car {

   public void accelerate() {
      //this acceleration is crazy!
   }
}
```

Maybe not…

Imagine most of our codebase was written with this

```
public interface Car {

   void turnOnEngine();
   void accelerate();
}

// lots of code

…. myCar.turnOnEngine();

// lots of code
```

What happens when we change to this?

```
public interface Car {

   void accelerate();
}
```

Is this better?

```
public interface Vehicle {

   void accelerate();
}
public interface Car extends Vehicle {

   void turnOnEngine();
}

public class MotorCar implements Car {
   private Engine engine;

 public void turnOnEngine() {
      //turn on the engine!
      engine.on();
   }
   public void accelerate() {
      //move forward!
      engine.powerOn(1000);
   }
}

public class ElectricCar implements Vehicle {
   public void accelerate() {
      //this acceleration is crazy!
   }
}
```

Interface Segregation Principle (ISP): Large interfaces should be split into smaller ones. Classes that implement the interface should only have to provide implementations of the methods relevant to them

Other ways to phrase this:

Clients should not be forced to depend upon interfaces they do not use

OR

Reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones

Analogous to single responsibility principle: If an interface has multiple purposes, then it (and its implementations!) is more likely to be affected by a requirements change. Break up into multiple interfaces, where each individual interface has only one purpose. Leads to more smaller interfaces.

```
public interface BearKeeper {
   void washTheBear();
   void feedTheBear();
   void petTheBear();
}
```

Does the bearkeeper at the zoo really need to pet the bears? That’s too dangerous...

Is this better?

```
public interface BearCleaner {
   void washTheBear();
}

public interface BearFeeder {
   void feedTheBear();
}

public interface BearPetter {
   void petTheBear();
}

public class BearCarer implements BearCleaner, BearFeeder {
   public void washTheBear() {
      //I think we missed a spot...
   }

   public void feedTheBear() {
      //Tuna Tuesdays...
   }
}

public class CrazyPerson implements BearPetter {
   public void petTheBear() {
      //Good luck with that!
   }
}
```

}

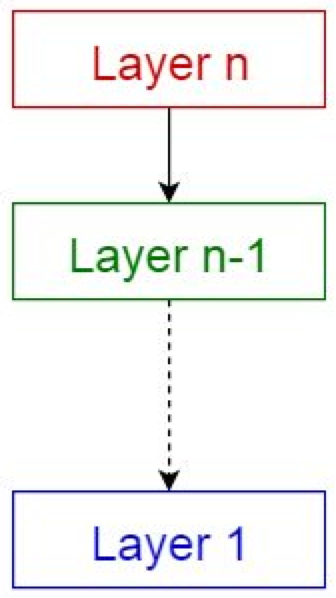
Let’s reconsider our motor and electric cars…

```
public interface Vehicle {

   void accelerate();
}
public interface GasolinePowered {

   void turnOnEngine();
}
public interface Car extends Vehicle, GasolinePowered;

public class MotorCar implements Car {
   private Engine engine;

 public void turnOnEngine() {
      //turn on the engine!
      engine.on();
   }
   public void accelerate() {
      //move forward!
      engine.powerOn(1000);
   }
}
public class ElectricCar implements Vehicle {
   public void accelerate() {
      //this acceleration is crazy!
   }
}
```

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules, which makes it hard to reuse the high-level modules. Instead both should depend on abstractions.

This seemingly contradicts the layered architectural style, where each layer builds on the layer below. The contradiction is resolved by depending on an abstraction of the layer below, not the details of the lower layer.

Abstractions should not depend on details. Details should depend on abstractions.



What is wrong with this?

```
public class VirtualPC {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public VirtualPC() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }
}
```

Is this better?

```
public interface Keyboard { }

public class StandardKeyboard implements Keyboard { }

public class VirtualPC {

    private final Keyboard keyboard;
    private final Monitor monitor;

    public VirtualPC(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

```
public class StringProcessor {
   private final StringReader stringReader;
   private final StringWriter stringWriter;

   public StringProcessor(StringReader stringReader, StringWriter
stringWriter) {
      this.stringReader = stringReader;
      this.stringWriter = stringWriter;
   }
   public void printString() {
      stringWriter.write(stringReader.getValue());
   }
}
```

Which are better design choices?

1. *StringReader* and *StringWriter* (the low-level components) are concrete classes placed in the same package. *StringProcessor* (the high-level component) is placed in a different package. *StringProcessor* depends on *StringReader* and *StringWriter*.

2. *StringReader* and *StringWriter* are interfaces placed in the same package along with their implementations. *StringProcessor* now depends on abstractions, but the low-level components don't.

3. *StringReader* and *StringWriter* are interfaces placed in the same package together with *StringProcessor*. Now, *StringProcessor* has the explicit ownership of the abstractions. *StringProcessor, StringReader,* and *StringWriter* all depend on abstractions.

4. *StringReader* and *StringWriter* are interfaces placed in a separate package from *StringProcessor*, and all depend on abstractions.

Another D principle (SOLIDD?)

Don’t Repeat Yourself principle (DRY): Avoid duplicate code and redundant work, instead abstract out common code and put in a single location. Make sure that you only implement each feature and requirement once

Do not copy/paste - if you find code that you would like to copy/paste (from your own codebase), remove from its original location and put in a new method, then both places should call that method

When similar code was written independently, detect and combine during refactoring

Some IDEs and refactoring tools automate “code clone” detection, but this only works if the code *looks* similar, not if it *behaves* similarly but looks different - but similarly behaving code requires redundant work whether or not it looks similar

DRY is about putting a piece of functionality in one place vs. SRP is about making sure a class does only one thing

Reminder:

The first assessment will be posted tonight/tomorrow Friday October 16 at 12:01am, and is due by Monday October 19 at 11:59pm. You can spend as little or as much time during that period as you like. Piazza will be closed during that period except for private messages to the instructor. Submit early, submit often, but do not submit late.

The assessment will be based on lectures (watch the videos!) and the individual assignments. If you haven't already, you should follow the links embedded in the lecture notes and assignments.

"Assignment T1: Preliminary Project Proposal" due October 27
https://courseworks2.columbia.edu/courses/104335/assignments/486922

Each team proposes their own project, within constraints:

- Must impose authenticated login.
- Must be demoable online (e.g., zoom or discord).
- Must store some application data persistently (database or key-value store).
- Must use some publicly available API beyond those that "come with" the platform. It does not need to be a REST API.

Describe Minimal Viable Product (MVP) both in prose - a few paragraphs - and via 3-5 "user stories".

< label >: As a < type of user >, I want < some goal > so that < some reason >.

My conditions of satisfaction are < list of common cases and special cases that must work >.

Describe acceptance testing plan that covers these cases.

List the tech you plan to use. If different members of the team plan to use different tools, please explain.

Two sample team projects from a previous offering of this course are linked below. All submitted assignments as well as the code are included in each repository.   These projects had three iterations because the team project ran the entire duration of the course, there was no individual project.

Code Phoenix - https://github.com/s4chin/coms_4156

Space Panthers - https://github.com/wixyFun/openSpace