Java Week 2 - SOLID: Handout

# Handout Objectives

Theoretical understanding of the SOLID Principles

# Single Responsibility Principle

The single responsibility principle is that each class should have a single responsibility. If a class has multiple responsibilities, it should be broken down into smaller components.

Classes should be broken down in a manner which will **decrease coupling** (decreasing interdependency) and **increase cohesion** (degree to which the elements inside a module belong together).

Low coupling and high cohesion indicate good quality code.

Classes which are very large are sometimes referred to as a “God” class. As the class can and does do everything. The inherent problem with this is that it is likely to change very frequently. Because it is so large, it is possible that one piece of code change, could affect parts we didn’t mean to change. This would cause a domino of changes to other classes it then interacts with.

Also having a class makes it harder to understand the direct purpose of the class and what its overall features should be.

When this happens, other developers are less likely to interact with the code to improve it, as they don’t know what it is meant to be doing.

This then ultimately leads to unmaintainable code.

**Single Responsibility Principle - Bad example of code**:

public class Car {

private String colour;

private String model;

private int distance;

Car() {

this.distance = 0;

}

public void setModel(String model) {

this.model = model;

}

public void sprayPaint(String colour) {

this.colour = colour

}

public void drive() {

this.distance += 1;

}

}

* Spray paint isn’t a behaviour of a car, it is a behaviour of a robot, or a spray painter.
* Inside the class, you only want to have behaviours which are of the Car.

**Single Responsibility Principle - Good example of code:**

public class Car {

private String colour;

private String model;

private int distance;

Car (String colour, String model, int distance) {

this.colour = colour;

this.model = model;

this.distance = distance;

}

public void drive() {

this.distance += 1;

}

}

Public class ToyotaFactory {

private String colour;

public void selectColour(String colour) {

this.colour = colour

}

public Car buildCar() {

return new Car(colour, “Toyota”, 0);

}

}

Here the building steps to create the car are put inside a ToyotaFactory class. This class then creates a Car. Changing wheels, replacing windscreens, repainting the car – these are all not behaviours of cars.

# Open/Closed Principle

The open closed principle states that classes should be **open for extension**, but **closed for modification**.

You want developers to extend and add to your functionality, without having to directly change the code. This is usually because our previous code has gone through testing, and we can be sure that it works!

Essentially, we should strive to write code that doesn’t have to be changed every time the requirements change.

This can be done in multiple different ways. The most common forms use **inheritance** and **polymorphism**.

**Open/Closed Principle – bad example of code:**

public int **getArea**( Shape shape) {

**If** (shape instance of Rectangle) {

**return** shape.height \* shape.width;

}

**else if** (shape instance of Traingle) {

**return** shape.height \* shape.width / 2;

}

**else if** (shape instance of Circle) {

**return** shape.height \* shape.height / 4 \* Math.PI;

}

}

This piece of logic can not be extended to other shapes. Therefore, it will most likely need to be changed later.

**Open/Closed Principle – good example of code:**

public void **getArea**(Shape shape) {

shape.calculateArea()

}

This function is a lot tidier and can work with any shape, without having to add extra logic each time.

public interface Shape {

public int calculataArea();

}

This interface means that every object which implements shape, will have a calculateArea() function. One example would be a Rectangle class:

public class Rectangle implements Shape {

private int length;

private int width;

public Rectangle (int length, int width) {

this.length = length;

this.width = width;

}

public int calculateArea() {

return length \* width;

}

}

We are not limited by the number of shapes we can have, therefore the getArea function is open for extension, yet closed to modification.

# Liskov’s Substitution Principle

Liskov’s substitution principle states functions that use pointers to base classes must be able to use objects of derived classes without knowing it.

**Liskov’s Substitution Principle - Bad example of code:**

**Bird**

- fly()

**Owl**

**Penguin**

public abstract class Bird {

public int altitude;

public abstract void fly();

}

public class Owl extends Bird {

public void fly() {

this.altitude += 1000;

}

}

public class Penguin extends Bird {

public void fly() {

throws new UnsupportedOperation();

}

}

public class AnimalKingdom {

public void learnToFly(Bird bird) {

if (bird.getClass==”Owl”) {

bird.fly();

}

}

}

The basetype Bird is not directly substitutable by derived type Penguin. Therefore in the AnimalKingdom class, we must first know which derived type we are using before running any of the Bird methods.

**Liskov’s Substitution Principle - Good example of code:**

First we reorder the code, so that derived types are directly substitutable.

**Bird**

**FlightyBird**

- fly()

**NonFlightyBird**

- flap()

**Owl**

**Penguin**

public abstract class Bird {

public abstract class Bird {}

public abstract class FlightyBird implements Bird {

public abstract fly();

}

public abstract class NonFlightyBird implements Bird {

public abstract flap();

}

public class Owl {

public int altitude;

public void fly() {

this.altitude += 1000;

}

}

Public class Penguin {

public int speed;

public void flap() {

this.speed += 5;

}

}

public class AnimalKingdom {

public void learnToFly(FlightyBird flightyBird) {

flightyBird.fly();

}

}

This time only the Owl is going to get the fly functionality. And Owl is the only class which can be passed through to the learnToFly method. Therefore no checks are needed!

# Interface Segregation Principle

The interface segregation principle states you shouldn’t be forced to depend on methods that will not use.

In other words, only have methods in an interface together if they all need to be implemented.

Take a SpreadSheetConversion interface for example.

This interface could have methods for ExcelToCSV(), CSVToExcel(), CSVtoPDF() and ExcelToPDF()

When we create a Converter class which implements this interface, we are going to have to override all these methods. However, I might only need the Excel-CSV functions, **not all of them**!

**Interface Segregation Principle – bad example of code:**

public interface SpreadSheetConversion {

public CSV ExcelToCSV();

public Excel CSVToExcel();

public PDF ExcelToPDF();

public PDF CSVToPDF();

}

public class Converter implements SpreadSheetConversion {

public CSV ExcelToCSV() {

// Conversion logic

}

public Excel CSVtoExcel() {

// Conversion logic

}

public PDF ExcelToPDF() throw UnsupportedOperation {

Throws new UnsupportedOperationException();

}

public PDF CSVToPDF() throw UnsupportedOperation {

Throws new UnsupportedOperationException();

}

}

The exception thrown isn’t good practice as it will lead to unmaintainable code.

**Interface Segregation Principle – good example of code:**

The solution is a rather simple one.

Break the SpreadSheetConversion interface into smaller interfaces!

public interface ExcelCSVConverter {

public CSV ExcelToCSV();

public Excel CSVToExcel();

}

public class Converter implements ExcelCSVConverter {

public CSV ExcelToCSV() {

// Conversion logic

}

public Excel CSVtoExcel() {

// Conversion logic

}

}

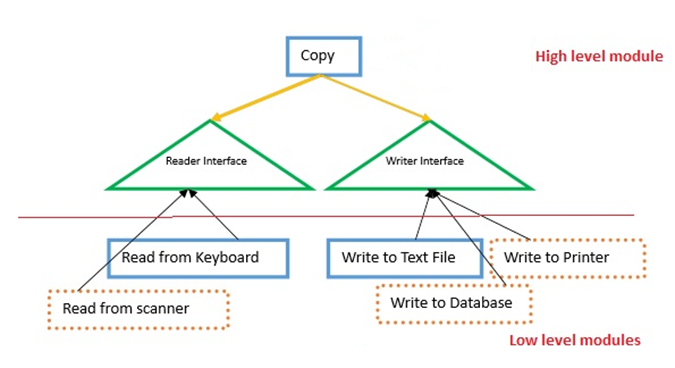
Segregating interfaces this way has a great advantage when it comes to readability and was as maintainability.

**Dependency Inversion Principle**

The dependency inversion principle has two key points.

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Point 1. Means that classes should be based on interfaces, regardless of whether they are high level modules or the low-level modules that they call.



In the figure above we are writing a function to copy something from one area to another area.

Copy does not call the ReadFromKeyboard class directly, as this would cause dependencies of high-level modules on the low-level modules modules.

We can containerise the interface concept with the implementation class. By doing so, we can include many different implementations (see Open/Close principle).

This leads onto point 2. Which in simply terms means that even if details change, the interface does not need to, for the concept has remained unchanged. Take a car. We can switch out the engine, change the brakes etc. But the way we interact with the car to drive has remained unchanged. The steer, gears and pedals are still there.