Writing SOLID Code

Thus far, we've written quite a bit of segregated code. Each file has a simple class that runs its code and the results are typically handed off to another file for further use. However, most code–especially OOP code–doesn't work this way. Generally speaking, the whole point behind OOP is to leverage the power of objects, as we've discussed with the Four Pillars of OOP.

With great power comes great responsibility, though. Keeping our code easy-to-read, reusable, and easily modifiable can represent some challenges. To help us overcome these challenges, Robert C. Martin (aka "Uncle Bob") developed a list of principles that eventually became SOLID: Single responsibility, Open/closed, Liskov integration, Interface segregation, and Dependency inversion. Let's go through them one at a time.

##1: Single Responsibility Principle

The Single Responsibility Principle states that "a class should have one, and only one, reason to change." Essentially, this means that each class should only have one job.

Don't make these, photo by Patrick on Unsplash

If you were creating a program to calculate the perimeters of various shapes, you could create a class called `shapePerimeterCalculator`. This class would likely take in a

type of shape and other parameters, like side lengths for rectangles and radii for circles–and that would all be fine. The shapePerimeterCalculator's job, after all, is to calculate the perimeter.

But what if you wanted to output the perimeter after you found it? You may think, "easy! System.out.println at the bottom of the shapePerimeterCalculator!" In a sense, this is true–but in the real world, we rarely put anything straight to the console. In reality, you may need the results of that information in many different places and/or in many different formats.

In order to stick to the Single Responsibility Principle, you'd want to make a printer class to print your calculated perimeter. It may seem silly when you think of just one class using this new printer class, but what if you wanted to expand your calculator? Maybe you'll do some area calculations, or 3D volumes and complete surface areas? Will you add a new print statement to each of these, or would you rather pass the information off to a dedicated printer? You could use your printer for error messages and anything else you needed to output, too. Best of all, if something breaks with your print statements, you'll have exactly one place to go to fix it, rather than three or more.

Make these, Photo by Elena Rouame on Unsplash

Consider also that you may need to output your information in other formats, send it to a database, or display it on a screen. By having separate classes handle separate cases, you can keep your logic tightly controlled.

The Single Responsibility Principle ensures that your code remains reusable and maintainable by keeping your classes focused on achieving one goal.

##2: Open/Closed Principle

Probably the most confusingly named of the SOLID Principles is the Open/Closed Principle. Let's take it in two parts:

1. Open: the open part of the Open/Closed Principle refers to the fact that a class should be open for extension. As you've seen throughout this week, extension means adding something on to a class, such as an attribute or a method. For instance, we may have a `Food` class that has an `isEdible` attribute, a `calorieCount`, etc., but we could add onto that class–extend it–with something like an `isArt` attribute. Most food won't be art–it'll just be food–but a class that's open to extension can be made into something *more* whenever we need it to be.

An artistic wedding cake, photo by Deva Williamson on Unsplash

2. Closed: the closed part of the Open/Closed Principle refers to modifying the original class. In the example above, we extended the class to make it something more than it was originally, but we didn't have to dig into the class itself and assign new attributes to the `Food` class. It's that simple.

To follow the Open/Closed Principle, you can use classes and interfaces to hand down the attributes and methods that all of your instances will need–and then extend those basics to take care of edge cases, like edible art.

The Open/Closed Principle means keeping your classes/interfaces open for extension, but making them foolproof enough that you don't have to go in and edit the underlying class/interface itself.

##3: Liskov Substitution Principle

Named after Barbara Liskov, this principle can be a bit hard to understand, especially when you read her original 1987 explanation:

What is wanted here is something like the following substitution property: if for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.

What it means in practice is fairly simple: any class derived from another should fulfill the base class's behaviour. Essentially, if you're going to define a `Bird` class that has a method called `fly()`, later defining an emu, an ostrich, or a penguin, would cause you to break the Liskov Substitution Principle. The base class said that instances of this class would have a `fly()` method, but these birds can't fly. It really is that simple: the instances of a class must represent all of the features of the parent class.

To follow the Liskov Substitution Principle, return to the Open/Closed Principle and think: what does the base class/interface need, and what should be an extension built upon that foundation? Keep the extensions separated, and you'll be fine.

##4: Interface Segregation Principle

The Interface Segregation Principle builds upon all of the above: interfaces should not pass on any behaviour to the classes that extend them beyond what those classes will use. If, instead of a `Bird` class, we had a `IBird` interface, we shouldn't have it feature an abstract method called `fly()` any more than it should have an abstract method called `swim()`. While most birds do fly, and many can swim, it doesn't make much sense to ensure that all birds inherit these methods because some of them won't be using them.

In practice, this means writing more granular interfaces, rather than trying to capture everything in one big one. As the name suggests, segregate your needed functionality into separate interfaces, then have your classes implement as many of them as they need.

The Interface Segregation Principle is exactly how it sounds: keep your interfaces separated, and ensure that all instances that inherit from them need the functionality that they provide.

##5: Dependency Inversion Principle

The Dependency Inversion Principle basically states that the code we write should depend on abstractions. In the words of Uncle Bob, "high level modules should not depend upon low level modules. Both should depend on abstractions," and "abstractions should not depend on details. Details should depend upon abstractions."

While it may be a little early in your dev career to be getting so abstract, this really should be your end goal: wherever possible, your code should rely on abstraction. Abstraction helps to keep your code loosely coupled, which basically means that no two parts of your code rely directly upon each other any more than they have to.

Suppose you had a class A that depended upon a class B, and that class B depended upon class C. If class C changes, say in the way that it outputs its response, class B would have to change. If class B has to change, there's a very high likelihood that class A would have to change, too. In companies that scale their products (like banks), this can be a major cause for concern.

However, if classes A, B, and C, all relied upon abstract methods from an interface or two, we could keep their interdependency to a minimum. If C changed, it wouldn't necessarily require that A or B be changed–and that's exactly what we want.

The Dependency Inversion Principle tells us to rely on abstractions, rather than concrete examples. Where possible, we should leverage the power of interfaces to ensure that changes we make in one part of our code don't cause cascading troubles for the rest of our program.

#Knowledge Check

In the following class, select all code that should be removed to another class/interface under the Single Responsibility Principle.

```
public class Book {
private String name;
private String author;
private String text;
public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public String getAuthor() {
return author;
}

public void setAuthor(String author) {
this.author = author;
}

public String getText() {
return text;
```

```
}

public void setText(String text) {
this.text = text;
}

void printTextToConsole(String text){
System.out.println(text);
}
}
```

Options

Single Responsibility Example

Removing functionality from a class that doesn't match what the class should be doing.

Open/Closed Example

Building upon a class without changing the code of the class itself.

Dependency Inversion Example

Creating a new interface to ensure that classes don't rely on each other.

Given that snakes exist, select all code that should be removed from this class in order to follow the Liskov Substitution Principle.

```
public class Reptile {
private int numberOfEyes;
private String name;

public void walk(String name) {
System.out.println(name + " can walk.");
}

public void layEggs(String name) {
System.out.println(name + " lays eggs.");
}

public void printEyes(String name, int numberOfEyes) {
System.out.println(name + " has " + numberofEyes "eyes!");
}
}
```

#Extra Resources

If you know some Javascript and want a very plain language explanation of SOLID, we recommend watching Kyle's series here. Even if you don't know Javascript, you'll probably be OK–it's far simpler than Java, and you'll be learning it later in the course anyway.

If you're ready for more complex examples that use Java, AmigosCode has a good video here. Be prepared, though: the examples do get fairly complex.