Programación 3

SOLID design principles

[These pages except when noted contain the code by Mike Knepper available at https://github.com/mikeknep/SOLID.]

This repository contains examples of the five SOLID design principles of object-oriented programming. The examples are written in Java. Each example has a "good" and "bad" version to demonstrate adherence to and violation of the principle, respectively.

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

1. Single Responsibility Principle

Every class should have a single responsibility. There should never be more than one reason for a class to change.

This example is derived from my Tic Tac Toe game. The bad example provides a generic Board class that does board-related things--it stores the values of spots on the board, returns the board's rows, and prints the board out to the screen. This approach reminds me of models in many Rails apps. On the surface, everything seems legitimately related to a real-world Board object, but the Single Responsibility Principle tells us that this class is actually handling far too many responsibilities.

Consider the Board class in the "good" example. The only thing it is responsible for is knowing the values of its spots. It is entirely unconcerned with how those spots are being manipulated per the rules of Tic Tac Toe (rows, columns, diagonals) or displayed to the user (in a console, on the web, etc.). The BoardShaper and BoardPresenter classes are similarly focused on specific tasks. They are also only passed attributes they need; for example, BoardShaper objects are initialized with only a size (they don't need the whole board).

Bad design

```
import java.util.ArrayList;

public class Board {
    ArrayList<String> spots;

    public Board() {
        this.spots = new ArrayList<String>();
        for (int i = 0; i < 9; i++) {
            this.spots.add(String.valueOf(i));
        }
    }

    public ArrayList<String> firstRow() {
        ArrayList<String> firstRow = new ArrayList<String>();
        firstRow.add(this.spots.get(0));
        firstRow.add(this.spots.get(1));
        firstRow.add(this.spots.get(2));
        return firstRow;
    }

    public ArrayList<String> secondRow() {
        ArrayList<String> secondRow = new ArrayList<String>();
        secondRow.add(this.spots.get(3));
        secondRow.add(this.spots.get(4));
        secondRow.add(this.spots.get(5));
        return secondRow;
    }

    public ArrayList<String> thirdRow() {
        ArrayList<String> thirdRow = new ArrayList<String>();
        thirdRow.add(this.spots.get(6));
        thirdRow.add(this.spots.get(7));
        thirdRow.add(this.spots.get(8));
        return thirdRow;
    }

    public void display() {
        String formattedFirstRow = this.spots.get(0) + " | " +
this.spots.get(1) + " | " + this.spots.get(2) + "\n" + this.spots.get(3) + "
| " + this.spots.get(4) + " | " + this.spots.get(5) + "\n" +
this.spots.get(6) + " | " + this.spots.get(7) + " | " + this.spots.get(8);
        System.out.print(formattedFirstRow);
    }
}
```

Good design

```
import java.util.ArrayList;

public class Board {
    int size;
    ArrayList<String> spots;

    public Board(int size) {
        this.size = size;
        this.spots = new ArrayList<String>();
        for (int i = 0; i < size; i++) {
            this.spots.add(String.valueOf(3*i));
            this.spots.add(String.valueOf(3*i + 1));
            this.spots.add(String.valueOf(3*i + 2));
        }
    }

    public ArrayList<String> valuesAt(ArrayList<Integer> indexes) {
        ArrayList<String> values = new ArrayList<String>();

        for (int index : indexes) {
            values.add(this.spots.get(index));
        }

        return values;
    }
}

public class BoardPresenter {
    Board board;

    public BoardPresenter(Board board) {
        this.board = board;
    }

    public void displayBoard() {
        String formattedBoard = "";
        for (int i = 0; i < this.board.size*this.board.size; i++) {
            String borderOrNewline = "";
            if ((i+1) % board.size == 0) {
                borderOrNewline += "\n";
            }
            else {
                borderOrNewline += "|";
            }
            formattedBoard += board.spots.get(i);
            formattedBoard += borderOrNewline;
        }
```

```
        System.out.print(formattedBoard);
    }
}

import java.util.ArrayList;

public class BoardShaper {
    int size;

    public BoardShaper(int size) {
        this.size = size;
    }


    public ArrayList<ArrayList<Integer>> rowIndexes() {
        ArrayList<ArrayList<Integer>> rowIndexes = new
ArrayList<ArrayList<Integer>>();

        for (int i = 0; i < this.size; i++) {
            ArrayList<Integer> row = new ArrayList<Integer>();
            for (int j = 0; j < this.size; j++) {
                row.add((i*size)+(j));
            }
            rowIndexes.add(row);
        }

        return rowIndexes;
    }
}
```

2. Open/Closed Principle

> **Software entitites should be open for extension, but closed for modification.**

I find the Strategy Pattern a great demonstration of the open/closed principle. Notice how in the bad example, any time we want to add a new style of greeting, we have to change the Greeter class to accept a new type of personality. We don't want to have to modify our existing, working code to add something new. Instead, as demonstrated in the good example, we have a high-level Greeter class that is instantiated with some Personality... we don't know which yet, just that it will be some object that implements the Personality interface. Now we can add new objects like FormalPersonality, CasualPersonality, and IntimatePersonality, and just make sure they correctly implement the Personality interface (in this case that means they must have a `greet()` method). The Greeter class is now open for future extension, while remaining closed for modification.

Bad design

```
public class Greeter {
    String formality;

    public String greet() {
        if (this.formality == "formal") {
            return "Good evening, sir.";
        }
        else if (this.formality == "casual") {
            return "Sup bro?";
        }
        else if (this.formality == "intimate") {
            return "Hello Darling!";
        }
        else {
            return "Hello.";
        }
    }

    public void setFormality(String formality) {
        this.formality = formality;
    }
}
```

Good design

```
public class CasualPersonality implements Personality {
    public String greet() {
        return "Sup bro?";
    }
}

public class FormalPersonality implements Personality {
    public String greet() {
        return "Good evening, sir.";
    }
}

public class Greeter {
    private Personality personality;

    public Greeter(Personality personality) {
        this.personality = personality;
        }
```

```
    public String greet() {
        return this.personality.greet();
    }
}

public class IntimatePersonality implements Personality {
    public String greet() {
        return "Hello Darling!";
    }
}

public interface Personality {
    public String greet();
}
```

3. Liskov Substitution Principle

> **Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**

[Examples of bad/good code by Mike Knepper intentionally omitted for Programación 3. In this section, “The Liskov Substitution Principle” by Robert C. Martin will be followed as available at https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf.]

Listing 3.1

```
void DrawShape(const Shape& s){
  if (typeid(s) == typeid(Square))
    DrawSquare(static_cast<Square&>(s));
  else if (typeid(s) == typeid(Circle))
    DrawCircle(static_cast<Circle&>(s));
}
```

Listing 3.2

```
class Rectangle {
public:
  void SetWidth(double w) {itsWidth=w;}
  void SetHeight(double h) {itsHeight=w;}
  double GetHeight() const {return itsHeight;}
```

```
  double GetWidth() const {return itsWidth;}
private:
  double itsWidth;
  double itsHeight;
};
```

Listing 3.3

```
void Square::SetWidth(double w){
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}

void Square::SetHeight(double h) {
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

Listing 3.4

```
Square s;
s.SetWidth(1); // Fortunately sets the height to 1 too.
s.SetHeight(2); // sets width and heigt to 2, good thing.
```

But consider the following function:

```
void f(Rectangle& r) {
  r.SetWidth(32); // calls Rectangle::SetWidth
}
```

Listing 3.5

```
class Rectangle {
public:
  virtual void SetWidth(double w) {itsWidth=w;}
  virtual void SetHeight(double h) {itsHeight=h;}
  double GetHeight() const {return itsHeight;}
  double GetWidth() const {return itsWidth;}
private:
  double itsHeight;
  double itsWidth;
};

class Square : public Rectangle {
public:
  virtual void SetWidth(double w);
```

```
  virtual void SetHeight(double h);
};

void Square::SetWidth(double w) {
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h) {
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

Listing 3.6

```
void g(Rectangle& r){
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

4. Interface Segregation Principle

> **Clients should not be forced to depend on interfaces they do not use.**

It's easy to get caught in a trap of naming interfaces or abstract classes after real-world things. The problem with this approach is two-fold: the collection of methods defined in the interface increase as one adds more and more functionality of the object to the code (a violation of the Single Responsibility Principle), and implementations of the interface start to require exceptions to the rules of the interface. Consider the bad example here. It may seem reasonable to create a Bird interface that outlines the basic features of birds--they can fly and they can shed their feathers. It works for plenty of birds (like an eagle), but then we want to add penguins to our code. The penguin is technically a bird, but if we set it to implement our Bird interface, we have to throw an exception for the `fly()` method. The penguin should not be forced to depend on an action it cannot perform.

Instead, make interfaces more abstract. It helps adhere to both the SRP (the interface is only responsible for one particular behavior) and this Interface Segregation Principle because specific objects (like eagles and penguins) only implement the functionality they need.

Bad design

```
public interface Bird {
```

```
    public void fly();
    public void molt();
}

public class Eagle implements Bird {
    String currentLocation;
    int numberOfFeathers;

    public Eagle(int initialFeatherCount) {
        this.numberOfFeathers = initialFeatherCount;
    }

    public void fly() {
        this.currentLocation = "in the air";
    }

    public void molt() {
        this.numberOfFeathers -= 1;
    }
}

public class Penguin implements Bird {
    String currentLocation;
    int numberOfFeathers;

    public Penguin(int initialFeatherCount) {
        this.numberOfFeathers = initialFeatherCount;
    }

    public void molt() {
        this.numberOfFeathers -= 1;
    }

    public void fly() {
        throw new UnsupportedOperationException();
    }

    public void swim() {
        this.currentLocation = "in the water";
    }
}
```

Good design

```
public class Eagle implements FlyingCreature, FeatheredCreature {
    String currentLocation;
    int numberOfFeathers;

    public Eagle(int initialNumberOfFeathers) {
```

```
        this.numberOfFeathers = initialNumberOfFeathers;
    }

    public void fly() {
        this.currentLocation = "in the air";
    }

    public void molt() {
        this.numberOfFeathers -= 1;
    }
}

public interface FeatheredCreature {
    public void molt();
}

public interface FlyingCreature {
    public void fly();
}

public class Penguin implements SwimmingCreature, FeatheredCreature {
    String currentLocation;
    int numberOfFeathers;

    public Penguin(int initialFeatherCount) {
        this.numberOfFeathers = initialFeatherCount;
    }

    public void swim() {
        this.currentLocation = "in the water";
    }

    public void molt() {
        this.numberOfFeathers -= 4;
    }
}

public interface SwimmingCreature {
    public void swim();
}
```

5. Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

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

The DIP is concerned with reusability. The high-level modules or interfaces of an application should only be describing the "general flow" of behavior. In some cases this may be considered "business logic". Meanwhile, the low-level modules are written in such a way to apply their concrete details to the abstraction. (The Adapter Pattern is a good example of DIP.)

In the bad example here, the WeatherTracker depends on the low-level details of the different notification systems (a phone, an emailer, etc.). These should instead be depending on some abstraction. The good example introduces this abstraction--a "Notifier" interface.

Bad design

```
public class Emailer {
    public String generateWeatherAlert(String weatherConditions) {
        String alert = "It is " + weatherConditions;
        return alert;
    }
}

public class Phone {
    public String generateWeatherAlert(String weatherConditions) {
        String alert = "It is " + weatherConditions;
        return alert;
    }
}

public class WeatherTracker {
    String currentConditions;
    Phone phone;
    Emailer emailer;

    public WeatherTracker() {
        phone = new Phone();
        emailer = new Emailer();
    }

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
        if (weatherDescription == "rainy") {
            String alert = phone.generateWeatherAlert(weatherDescription);
            System.out.print(alert);
        }
```

```
        if (weatherDescription == "sunny") {
            String alert = emailer.generateWeatherAlert(weatherDescription);
            System.out.print(alert);
        }
    }
}
```

Good design

```
public class EmailClient implements Notifier {
    public void alertWeatherConditions(String weatherConditions) {
        if (weatherConditions == "sunny");
            System.out.print("It is sunny");
    }
}

public class MobileDevice implements Notifier {
    public void alertWeatherConditions(String weatherConditions) {
        if (weatherConditions == "rainy")
            System.out.print("It is rainy");
    }
}

interface Notifier {
    public void alertWeatherConditions(String weatherConditions);
}

public class WeatherTracker {
    String currentConditions;

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
    }

    public void notify(Notifier notifier) {
        notifier.alertWeatherConditions(currentConditions);
    }
}
```