Programación 3

U.D. 11 - Principios fundamentales del paradigma OO.

Principios de diseño SOLID

[Excepto donde se indique, estas páginas contienen código de Mike Knepper disponible en https://github.com/mikeknep/SOLID]

Este repositorio contiene ejemplos de los cinco principios de diseño orientado a objetos SOLID, escritos en Java. Cada ejemplo tiene una versión “buena” y otra “mala”, para demostrar el cumplimiento o incumplimiento del principio, respectivamente.

- Single Responsibility Principle (Principio de Responsabilidad Unica)
- Open/Closed Principle (Principio Abierto/Cerrado)
- Liskov Substitution Principle (Principio de sustitución de Liskov)
- Interface Segregation Principle (Principio de segregación de interfaz)
- Dependency Inversion Principle (Principio de la inversión de dependencias)

1. Principio de responsabilidad única (PRU)

Cada clase debe tener una única responsabilidad. Nunca debe existir más de una razón para que una clase cambie.

El ejemplo se basa en el juego de las tres-en-raya. En el ejemplo ‘malo’, Board (tablero) hace cosas relacionadas con un tablero: guardar los valores de las casillas, devolver las filas, imprimirlo. Parece correcto pero incumple el PRU: demasiadas responsabilidades.

Mira ahora la clase Board en el ejemplo ‘bueno’. Sólo es responsable de conocer los valores de sus casillas. No sabe nada acerca de las reglas del 3-en-raya o cómo se muestra el tablero. BoardShaper y BoardPresenter se ocupan de estas dos tareas.

Mal diseño

```
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);
    }
}
```

Buen diseño

```
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. Principio Abierto/Cerrado

Las entidades software (clases, paquetes, métodos, etc.) deben estar abiertas a su extensión, pero cerradas a su modifcación.

Fíjate como en el ejemplo malo, cada vez que queremos añadir un nuevo tipo de saludo, hemos de modificar la clase Greeter para que pueda tratar un nuevo tipo de personalidad. No queremos modificar código que ya funciona para añadir algo nuevo. En lugar de eso como se ve en el ejemplo bueno, Greeter funciona con cualquier tipo de personalidad, que representamos mediante el interfaz Personality. La clase Greeter está ahora abierta a la extensión, pero cerrada a la modificación (en lo que a personalidades respecta).

Mal diseño

```
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;
    }
}
```

Buen diseño

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

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


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();
    }
}
```

3. Principio de Sustitución de Liskov

> **Los métodos que usan referencias a superclase deben ser capaces de utilizar objetos de subclases sin saberlo**

[En esta sección no seguiremos los ejemplos de Mike Knepper, sino los de “The Liskov Substitution Principle” by Robert C. Martin : https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf.]

Listing 3.1 (mal diseño)

```
void DrawShape(Shape s){
  if (s instanceof Square)
    DrawSquare((Square)s);
  else if (s instanceof Circle)
    DrawCircle((Circle)s);
}
```

Listing 3.2

```
class Rectangle {
  public void setWidth(double w) {itsWidth=w;}
  public void setHeight(double h) {itsHeight=h;}
  public double getHeight() {return itsHeight;}
  public double getWidth() const {return itsWidth;}

  private double itsWidth;
  private double itsHeight;
}
```

Listing 3.3

```
class Square extends Rectangle {

  @Override public void setWidth(double w) {
    super.setWidth(w);
    super.setHeight(w);
  }

  @Override public void setHeight(double h) {
    super.setHeight(h);
    super.setWidth(h);
  }
}
```

Listing 3.4

```
Square s = new Square();
s.setWidth(1); // Afortunadamente, también inicializa la altura a 1.
s.setHeight(2); // inicializa anchura y altura a 2, buena cosa.
```

Listing 3.5

```
// código cliente: prueba unitaria
void testSetters(Rectangle r){
  r.setWidth(5);
  assertEquals("Width", r.getWidth(), 5);
  r.setHeight(4);
  assertEquals("Area", r.getWidth() * r.getHeight()), 20);
}
```

4. Principio de Segregación de Interfaz

El código cliente no debe ser forzado a depender de interfaces que no utiliza.

Considera el ejemplo malo aquí abajo. Parece razonable crear el interfaz Bird con el comportamiento básico de un pájaro (volar y mudar sus plumas). Funciona para un montón de pájaros, pero de repente necesitamos añadir pingüinos. Técnicamente es un pájaro, pero tendremos que implementar fly() haciendo que lance una excepción. El pingüino no puede ser forzado a depender de una acción que no puede ejecutar.

En lugar de eso, hagamos los interfaces más abstractos, ‘segregando’ el comportamiento de un interfaz en varios, de manera que objetos específicos sólo implementen los métodos que necesitan.

Mal diseño

```
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";
    }
}
```

Buen diseño

```
public interface FeatheredCreature {
    public void molt();
}

public interface FlyingCreature {
    public void fly();
}

public interface SwimmingCreature {
    public void swim();
}

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 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;
    }
}
```

5. Principio de Inversión de Dependencias (PID)

1. Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de abstracciones.

2. Las abstracciones no deben depender de detalles específicos. Éstos deben depender de las abstracciones.

En el ejemplo malo, WeatherTracker depende de los detalles de bajo nivel de los distintos sistemas de notificación (por teléfono, por email, etc). Estos deberían depender de alguna abstracción. En el ejemplo bueno, se introduce esa abstracción: el interfaz 'Notifier'.

Mal diseño

```
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);
        }
    }
}
```

Buen diseño (ligeramente modificado)

```
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;
    List<Notifier> notifiers;

    public WeatherTracker() {
      notifiers = new ArrayList<>();
    }

    public void registerNotifier(Notifier n) {
       notifiers.add(n);
    }

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

    public void notifyEveryone() {
      for (Notifier n : notifiers)
        n.alertWeatherConditions(currentConditions);
    }
}
```