Object Oriented Design Principles

The are some principles that we can use to deliver a better and more maintainable project.

SOLID principle for object-oriented design (OOD)

For a project that makes use of Object Oriented Design, please make use of SOLID principles as described as follow:

1. Single responsibility principle
2. Open for extensions Closed for mutations.
3. Liskov substitution.
4. Interface segregation.
5. Dependency Inversion

Single Responsibility Principle

> **Definition**
> A class must have one and one purpose only.

Every time you create a class or entity, you have to think yourself, what does this class do? If it has AND to what it can do, that means your class breaks the Single Responsibility Principle. Single responsibility doesn't mean that you should only have one method, no. You can have many ways, but those methods should serve the final same and only one purpose.

Sample code

```
public class UserSettingService
{
  public void changeEmail(User user)
  {
    if (checkAccess(user))
    {
       // Grant option to change
    }
  }

  public boolean checkAccess(User user)
  {
    // Verify if the user is valid.
  }
}
```

Do you see the problem with the code above? The problem is UserSettingService has multiple responsibilities, and one changes email for the user and the other is to checkAccess security. When ideally, you should split these concerns into two different classes as follows:

Sample code

```
public class UserSettingService
{
  public void changeEmail(User user)
  {
    if (SecurityService.checkAccess(user))
    {
       // Grant option to change
    }
  }
}

public class SecurityService
{
  public static boolean checkAccess(User user)
  {
    //check the access.
  }
}
```

Now the code above is conformant to the SRP since one Class only serves one purpose.

Open Closed Principle

Definition
Entities must be open for extension but close for mutations/modifications.

When one is creating an entity, be it model, classes, etc. One should make it extendable without changing the object that we are creating.

Sample code

```
class Shape {
 public double width;
    public double height;
 public double radius;

    Shape(double width, double height) {
        this.width = width;
        this.height = height;
    }

    Shape(double radius) {
  this.radius = radius;
 }

    public double getRectangleArea() {
        return width * height;
    }

    public double getCircleArea() {
  return this.radius * this.radius * Math.PI;
    }
}

public class Main {

    public static void main(String[] args) {
     Shape circle = new Shape(5);
     Shape rect = new Shape(3, 2);

        System.out.println(circle.getRectangleArea());
        System.out.println(rect.getCircleArea(rect));
    }
}
```

Do you see the problem with the code above? In the class Shape, it has two methods to handle rectangle and circle shapes, so when we introduce a new entity, then we need to mutate this class by adding more shape codes. And this is against the Open Closed Principle. Have a look at the code below

Sample Code

```
interface Shape {
    double getArea();
}

class Rectangle implements Shape {
    public double width;
    public double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }
}


class Circle implements Shape {
    public double radius;

    Circle(int radius) {
        this.radius = radius;
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}

public class Main {

    public static void main(String[] args) {
     Circle circle = new Circle(5);
     Rectangle rect = new Rectangle(3, 2);

        System.out.println(circle.getArea());
        System.out.println(rect.getArea());
    }
}
```

The above codes, now conformant to the Open-Closed Principle, where the Shape class is open for an extension for as much as shape as we want without the need for mutations to create a new Shape like a hexagon, etc.

Liskov's Substitution Principle

Definition
An entity should be replaceable by its descendant object.

The common problem that the Liskov Substitution principle can solve is the Rectangle-Square problem.

Let us assume that we have a Rectangle class like the following:

Sample code

```
public class Rectangle
{
  public void setWidth(int width)
  {
 this.width = width;
  }

  public boolean setHeight(int height)
  {
 this.height = height;
  }

  public void area()
  {
 return height * width;
  }
  ...
}
```

Now, let say we want to derive this Rectangle class into Square class implementation like the following:

Sample code

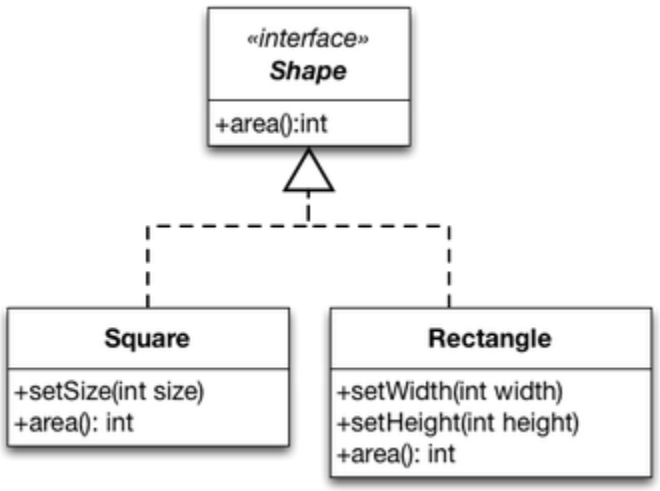
```
public class Square extends Rectangle
{
  public void setWidth(int width)
  {
 super.setWidth(with);
 super.setHeight(width);
  }

  public boolean setHeight(int height)
  {
 super.setWidth(width);
 super.setHeight(height);
  }
  ...
}
```

The problem we see from this design is that the Square implementation does conform to the mathematical properties of a rectangle where it has width and height, only that the width and the height are the same.

But, it breaks the behavior of the Square, because a square should not have width and height, it only has edges, which is the same for all the sides. The design of the rectangle and square breaks the Liskov substitution principle because Square is behaviorally not a correct substitution for Rectangle since it has different properties than Rectangle.

In conclusion, Square does not comply with the behavior of a rectangle.

To fix this problem, we might redesign it with the following:



Now, square and rectangle conform to the Shape interface, where both have the same universal property that is an area. Now, a square can have different features than a Rectangle, that is the size. So this design will comply with the Liskov substitution principle, where square can replace a shape. Also, a rectangle can be put in place of a square just fine.

Interface Segregation Principle

Definition
A client should only use the methods it needs.

If we take spiderman's term, with great power comes great responsibility. This term fits the ISP (Interface Segregation Principle), where a developer should never give a client a power that it does not use.

In layman terms, if an interface becomes so big, we split it into smaller interfaces, so that the derived implementation will only derive the methods it's interested in. So this will make sure all interface methods are implemented.

Take a look at the following example:

Sample code

```
interface Stream
{
  void read();
  void write();
  void trim();
  ...
}
```

In the above, we have three methods that all clients need to implement. But, not all clients need all three methods. So, there will be a client that only needs to read the stream but not the write or trim method. So, this design breaks the ISP.

To fix this problem, we should segregate the interface into smaller interfaces like the following:

Sample code

```
interface ReadStream
{
  void read();
  ...
}

interface WriteStream
{
  void write();
  ...
}

interface TrimStream
{
  void trim();
  ...
}
```

Dependency Inversion Principle

Definition
A boss should not be doing a low-level job.

What this principle does is that a high-level module should not be doing a low-level module job. Both should depend on abstractions.

Take a look at the following example:

Sample code

```
class ReadStream
{
  void read() {}
}

class WriteStream
{
  void write() {}
}

class MemoryStream
{
  private ReadStream _readStream = new ReadStream();
  private WriteStream _writeStream = new WriteStream();

  void flush()
  {
 _readStream.read();
 _writeStream.write();
  }
}
```

Looking at the example above, we see a problem. A MemoryStream class is a high-level class, but it does two level jobs, that is read and write to the stream. This design is against the DIP (Dependency Inversion Principle).

To fix this problem we can refactor with the following:

Sample code

```
interface Stream
{
  void flush();
}

class ReadStream implements Stream
{
  void flush()
  {
 read();
  }
  void read() {}
}

class WriteStream implements Stream
{
  void flush()
  {
 write();
  }
  void write() {}
}

class MemoryStream
{
  private List<Stream> _streams;

  MemoryStream(List<Stream> streams)
  {
 this._streams = streams;
  }

  void flush()
  {
 this._streams.forEach(s -> s.flush());
  }
}
```

Now, the high-level class will no longer have to do the low-level task but using the interface abstraction like above.

Related articles

- Object Oriented Design Principles
- Engineering Practices
- Setup for Bulk SMS Blast