Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Computer Science

Object-Oriented Programming

Laboratory work #3

SOLID principles

Authors:
Daniel Surdu

Supervisor:
Anastasia Șerșun

Chișinău - 2018

1 Work purpose:

The study and implementation of the two of the SOLID object-oriented design principles (L and D).

2 Task implementation

2.1 Liskov Substitution Principle

The principle states that you should be able to replace any instances of a parent class with an instance of one of its children without creating any unexpected or incorrect behaviors.

For example, in the code below, I have a base class rectangle and a derived class Square, we all now that a square is a rectangle with the same length of all edges and here is implemented the same logic.

```ruby
class Rectangle
  attr_accessor :x, :y

  def initialize(length = Random.rand(1..20), height = Random.rand(1..20))
    @x = length
    @y = height
  end

  def get_perimeter
    @x * 2 + @y * 2
  end



  def get_area
    @x * @y
  end

  def uniform_enlargement(percent)
    @x += @x * percent / 100.0
    @y += @y * percent / 100.0
  end

  def output_edges
    puts "a = #{@x}", "b = #{@y}", "c = #{@x}", "z = #{@y}"
  end
end
```

```ruby

class Square < Rectangle

  def initialize(size = Random.rand(1..20))
    @x = size
    @y = @x
  end
end
```

In this case the square implements all the rectangle functions the same and the program will work with no problems and will never create unexpected or incorrect behavior.

2.2 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) suggests that "High-level modules should not depend on low-level modules. Both modules should depend on abstractions. In addition, abstractions should not depend on details. Details depend on abstractions."

I created a class FigureGenerator that has a method called generate(fig = Square). In this method is shown the Dependency Inversion Principle because this method doesn't depend on the class that it uses. And in the code bellow is shown why is this a good practice.

```
require './square.rb'
require './rectangle.rb'
require './triangle.rb'
require './figure_generator'

generated_figure = FigureGenerator.new

[Triangle, Rectangle, Square].each do |figure|

  puts figure

  generated_figure.generate(figure)

  generated_figure.show_perimeter
  generated_figure.show_area
  generated_figure.output_figure
end
```

You can see here that I can use 3 different classes for the same method and everything will work just fine.

And on the other hand that method could look like this:

```
def generate()
  @figure = Square.new
end
```

In this case if something was to change in the Square class that could add a lot of errors and you will have to change the method not just the passed class. And this also make your class more expandable.

Conclusion

During this laboratory work, I studied two out of five SOLID principles: Liskov Substitution Principle and Dependency Inversion Principle. This enabled me to get a better understanding of the principles and to apply this knowledge in making my code better.

References

1 SOLID Principles, `https://robots.thoughtbot.com/back-to-basics-solid`

2 5 SOLID Principles, `https://rubygarage.org/blog/solid-principles-of-ood`