CentraleSupélec

Object Oriented Software Engineering

Practical 06

Paolo Ballarini

Learning outcomes:

- Application of design patterns
- Factory pattern
- Strategy pattern

Exercise 1. Extendable `Shape` classes

Design a class library for representing geometrical shapes. Initially your library should provide classes for representing circles, squares, rectangles and triangles objects (see the detailed specifications below for each of such classes). **Requirement**: the shape Library should be open to further extensions (it should be possible to easily add new shape classes later on).

Q1) Develop a UML class diagram which represents the shape library design.

Q2) Think about the patterns you are going to apply and whether it is useful to add a Factory

Q3) Using the JUnit framework develop some Unit Tests for testing the main functionalities of the Shape library. For example one of your test must define 10 shapes objects (at least one of each type, i.e. Triangle, Rectangle, Circle). Write the Unit Tests incrementally: each time you add a new class or functionality.

Roadmap to the solution.
Define the below described classes following the specifications given. For each class you should adopt the encapsulation principle. Each class should be equipped with getters (and setters if necessary) methods, as well as a coherent overriding of the `equals()` and `hashCode()` methods. You should define JUnit test cases for the main methods of each class (i.e. to test constructors that may throw an exception, and to test all methods declared in the `Shape` interface, i.e. the `area()`, `perimeter()`, `minSide()` and `maxSide()` methods.

class Point

- a Point is characterised by 2 real-valued variables representing its co-ordinates
- define a 2 arguments constructor for class Point, the first argument being the X-coordinate, the second being the Y-coordinate.

interface Shape: define an interface named `Shape` that declares the following methods

- `area()`: returns the value of the area of a Shape object
- `perimeter()`: returns the value of the perimeter of a Shape object
- `maxSide()`: returns the value of the largest size of a Shape object (for Circle that is assumed to be its diameter)
- `minSide()`: returns the value of the smallest size of a Shape object (for Circle that is assumed to be its radius)

class Rectangle

- a Rectangle is characterised by the co-ordinates of one of its vertices plus the length of its width and height
- define a 3 arguments constructor for class Rectangle, the first argument being the co-ordinates of one vertices the second and third arguments being the width and height of the rectangle
- **BadShapeCreationException**: the constructor of class Rectangle should throw a `BadShapeCreationException` to rule out any misuse attempt, e.g. attempting to create an Rectangle with equal width and height, or with non positive width and height.

class Square

- a Rectangle is characterised by the co-ordinates of one of its vertices plus the length of its edges.
- define a 2 arguments constructor for class Square, the first argument being the co-ordinates of one vertices the second being the size of its (four equally sized) edges.
- **BadShapeCreationException**: the constructor of class Square should throw a `BadShapeCreationException` to rule out any misuse attempt, e.g. attempting to create a Square with non positive length of its edges.

class Triangle

- a Triangle is characterised by the co-ordinates of its 3 vertices.
- define a 3 arguments constructor for class Triangle, the 3 arguments being the co-ordinates of its 3 vertices.
- **BadShapeCreationException**: the constructor of class Triangle should throw a `BadShapeCreationException` to rule out any misuse attempt, e.g. attempting to create a Triangle with non-distinct vertices.

class Circle

- a Circle is characterised by the co-ordinates of its center and the length of its radius.
- define a 2 arguments constructor for class Triangle, the 2 arguments being the co-ordinates of the center and the length of the radius.
- **BadShapeCreationException**: the constructor of class Circle should throw a `BadShapeCreationException` to rule out any misuse attempt, e.g. attempting to create a Circle with non-positive radius.

Exercise 2. Adding different sorting strategy to `Shape` class

Extend the shape library project (developed in Exercise 1) so to equip the `Shape` library with *interchangeable* sorting functionality for shape objects. In particular a client application that uses a collection of shape objects should be capable of sorting them according to the following criteria: *longest perimeter*, *larger area longest side shortest side*. The customer that commissioned you with the realisation of the `Shape` library requires that itshould also be extensible with respect to the sorting criteria, so that in future one can easily add new criteria for sorting the shape objects and the client application can easily swap from one sorting algorithm to another (i.e. what kind of design pattern would fit this kind of problem?).

Q1) extend the UML class diagram which represents the shape library design according to the new requirements.

Q2) modify the `Shape` class so that it realises a Strategy pattern which provides the (*longest perimeter*, *larger area longest side shortest side* sorting functionality (i.e. with respect to the *longest/shortest side* you should consider the "side" of a circle as its diameter, whereas for the *longest perimeter* you should consider the circumference of a circle).

Q3) Test you library and extend the `ShapeClient` application such that it displays also the result of sorting them out with all of the different criteria, hence display the list of shapes ordered with respect to the larger area, the longest perimeter, the longest side, and the shortest side.