**Portfolio 1 – Software Development**

**Victor Nielsen – Martin Bern**

**https://github.com/Victornrib/Portfolio1-SoftwareDevelopment**

**Introduction:**

For the first portfolio of the semester, our group will work in two different situations related to object-oriented programming and software engineering. Both of them will focus on how to implement classical OOP models and understand and apply software requirements.

The portfolio will be split in two main parts, in which each of them will target different situations:

* The first part will be based in build all the necessary environment and plan all the necessary tasks to create a small shapes model, which will contain basic forms as rectangles, triangles and circles; but also being designed with extensibility so other shapes could be added in the future without changing the basic model. For that, the first part will focus on:
* Establishing a development environment (in this case, using Git, Github, JAVA and InteliJ)
* Designing the application (using UML from building use case diagrams to static and dynamic models of the app)
* Planning the development of the application (using Trello – which is Kanban based)
* The second part will be the implementation of the model itself. It will have a common behavior, so that each shape instantiated can return:
* Its center
* Its area
* Its circumference
* Indicate whether or not a point is inside of it
* Compute the Euclidean distance from its center to the center of another shape

The design of the shapes will be made in UML and implemented in JAVA.

Resuming, the second part will be made of:

* An overall description of the shape model
* UML diagrams documenting representation and behavior of the shape model
* An implementation of the shape system
* Unity tests verifying behavior for all shapes

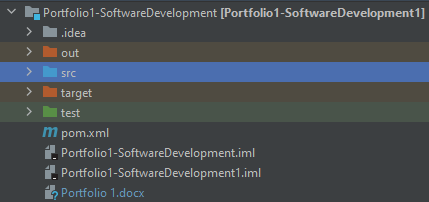
\*Some changes were made in the code from me (Victor Nielsen) to correct the Shape abstract class and also its diagram

**First part (Environment):**

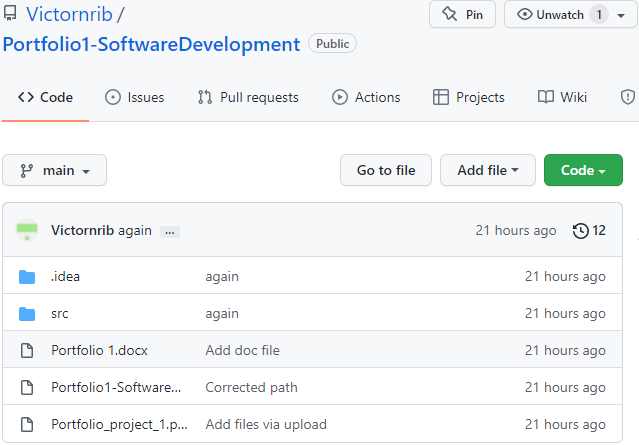
**Development environment:**

To implement a shapes model, based on the specifications given by the portfolio, we first started our development environment.

For that, we started creating a project in InteliJ using Oracle OpenJDK version 11.0.11, Maven and JUnit 5.8.2.



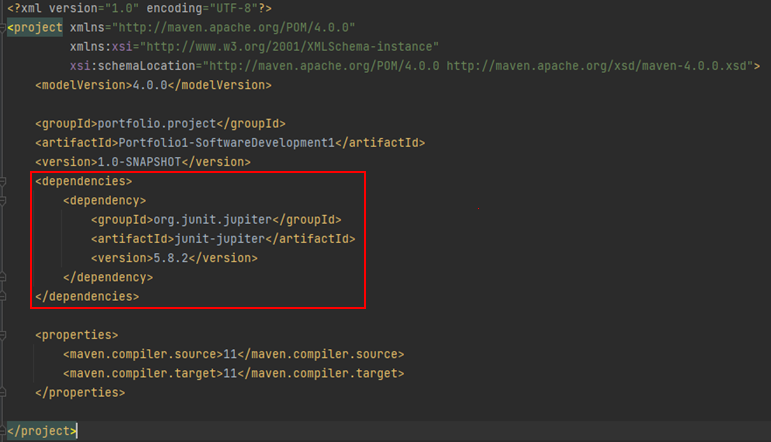
After that, we enabled Git in the project and exported it to a Github repository.



We did a remote repository because it gave us the ability to update our application from any device, so we could both make commits to the project. Also, it allowed us the search through Git History in the case of committing a version not working properly.

We also after created a branch “view” to develop all tasks related to a future implementation of a GUI for the user.

After we configured our pom.xml file, so that it could allow the use of the Junit 5, which would be crucial to test the class of the project.



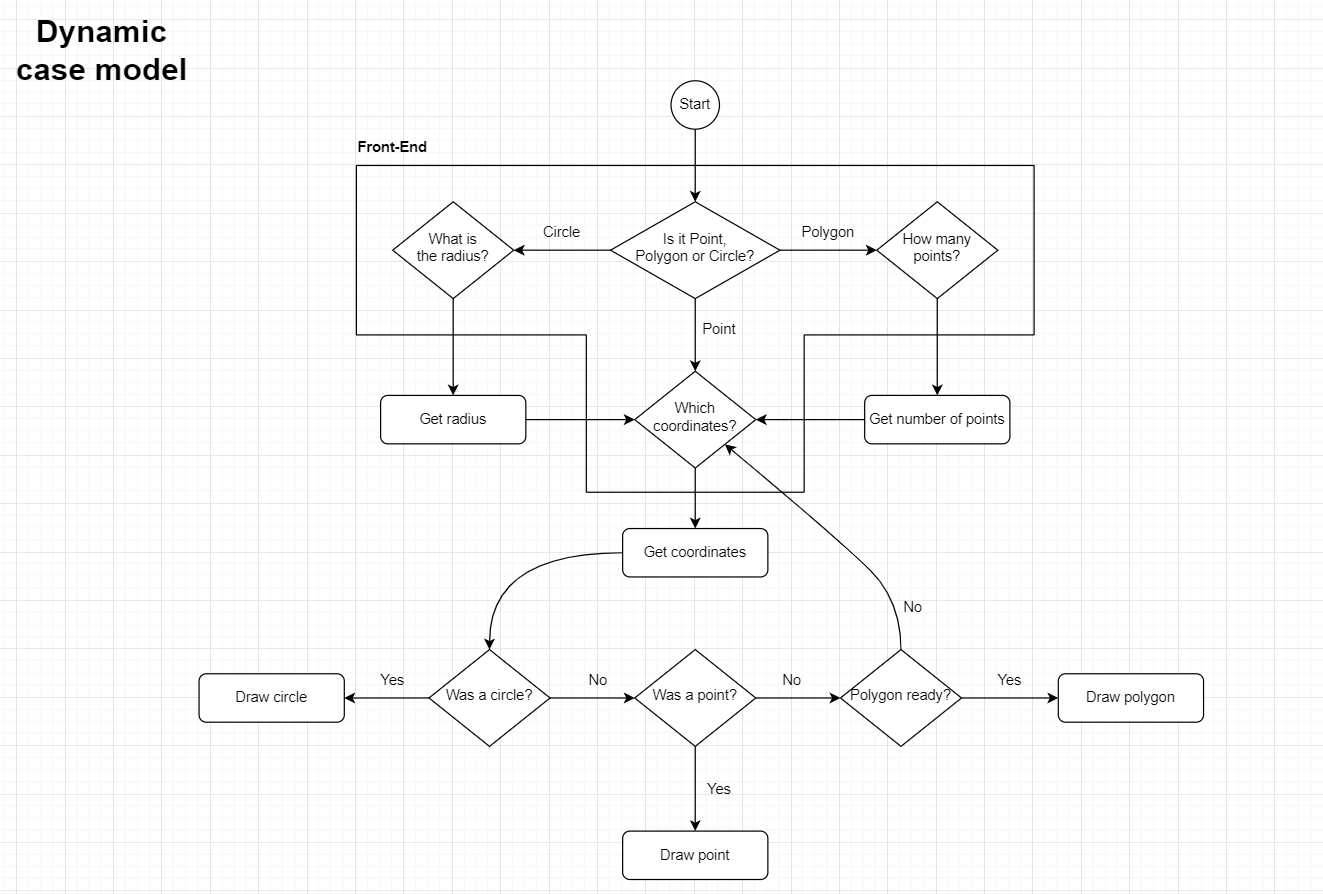
**Design of the application:**

**Designing a dynamic case model:**

For the app design, we started doing a dynamic case model with an Activity Diagram.

We wanted to make a model that could give us the general idea of the software and its interaction with the user.

The model would be a very useful tool to help to build the controller package, which would contain the main function for running the program.



Since we wanted to do an application with good potential to expand with new features, we only limited ourselves to three types of drawings: “Points”, “Polygons” and “Circles”.

They were chosen that way because they would have different internal representations. And also, in the case of the polygons, they would not be restricted to pre-defined forms, like squares or triangles, but to any convex polygon with 3 or more vertices.

Since both parts cover the same application, a more in depth look into the internal representations of each one of the base classes will be showed in our Class diagram in the second part of the portfolio. (Page 6)

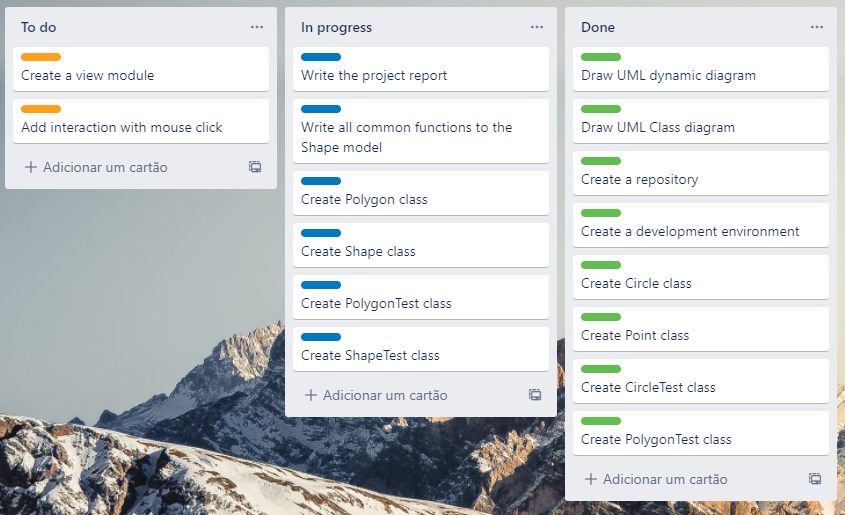
**Design pattern:**

Our design pattern chosen was the MVC (“Model – View – Controller”). We chose that one because it fits really well with the type of application that we want to develop.

* The model package contain all the necessary base classes for the Shape model, having their own variables and functions. This strategy is very good to encapsulate what is only necessary for the base classes and for the rest of the application, helping the development in the long term.
* After there is the controller package, which is responsible for being the middle ground between the Shape model and the user. He is the one who instantiate all the base classes and access their values.
* And for the last, there will be also a view package; responsible for displaying to the user the classes instantiated by the controller. This package is not implemented yet, since for now the assignment only required to have the base classes. (The view will be able to operate via classes extending JFrames and JPanels, creating an UI for the user)

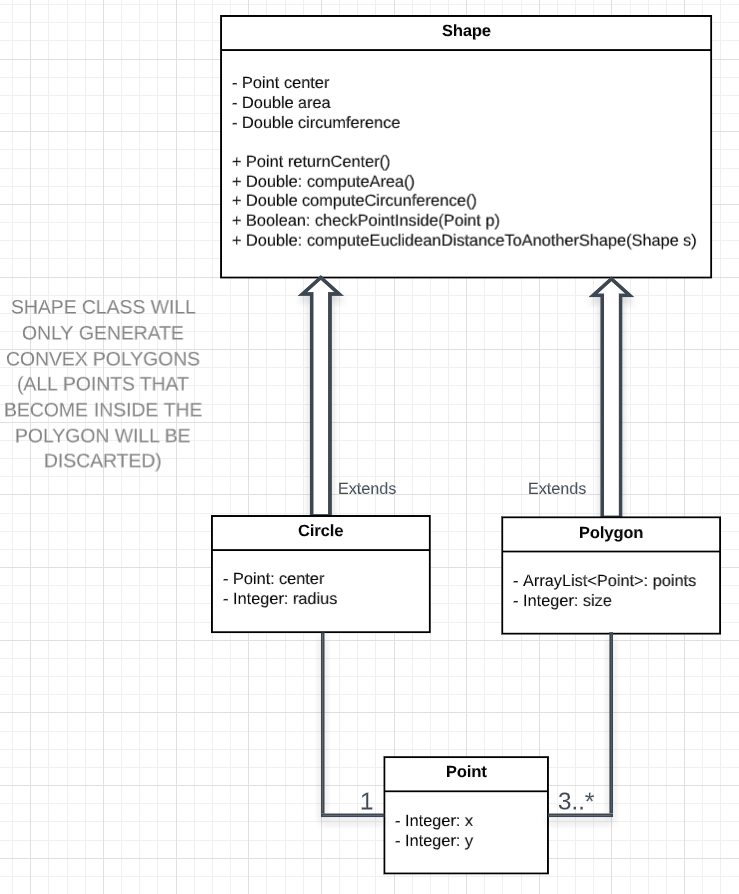
**Kanban based scheduler tool:**

With the intuition of organizing tasks, we also used Trello in our development. It was very useful to keep track of what were the next steps every time. It made us break a very complex program into many smaller problems that were feasible.



**Second part (Shape model):**

**Designing and explaining the model:**



The idea with our UML Class diagram was to show the relations of existence and hierarchy in our classes.

As it showed, the main class “Shape” is an abstract class, which means that it will never be instantiated directly by the controller. But even then, if the controller decides to instantiate, whether a Circle or a Polygon, both will have all the methods and variables inherited from the Shape class.

This saves a lot of time and lines of code since takes out the necessity of having the same variables and functions with classes with common behaviors.

The Point Class is not inherited from any class, but is crucial for the application, since all of them use points to represent some of its aspects.

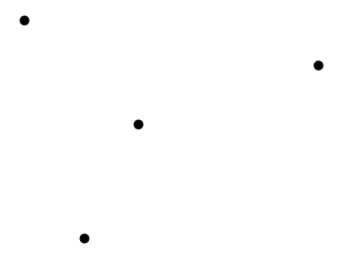
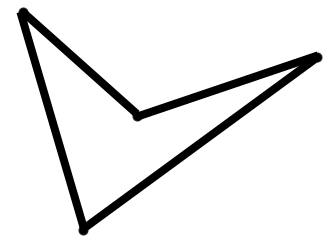
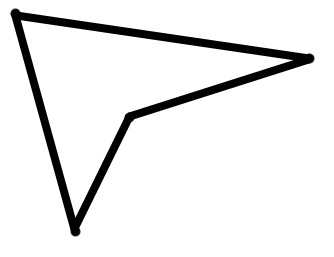
In the case of the Circle Class, only 1 point is necessary. He will correspond as the center where the circle will be located.

In the case of the Polygon Class, at least three or more points are necessary to generate it, since it has to be a two dimensional figure. Also, all the points must be in different coordinates in the plane, otherwise they would overlap, breaking the internal functions from the Shape class.

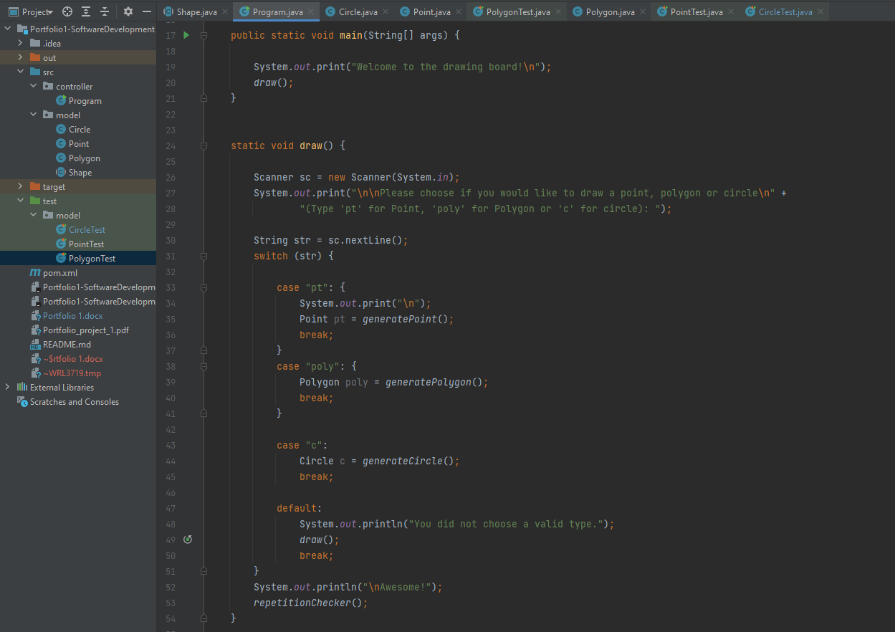
* Another feature, that will be implemented in the future will be the gift-wrapping algorithm for building convex polygons.

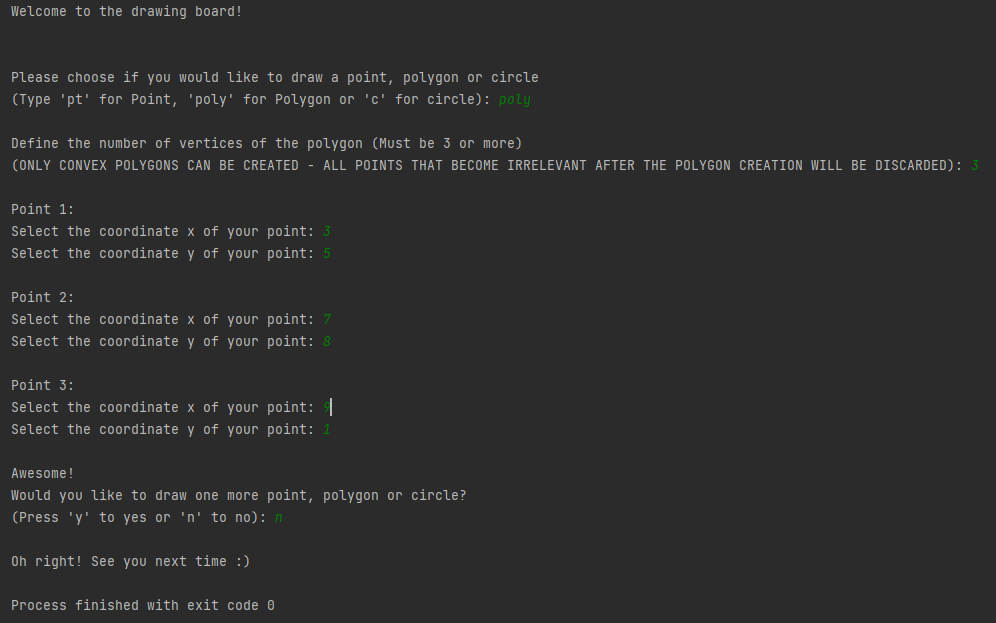
In the current state, the polygon that is generated by a set of points is the same representation and order that they are given to the Polygon constructor (given they are 3 or more different points).

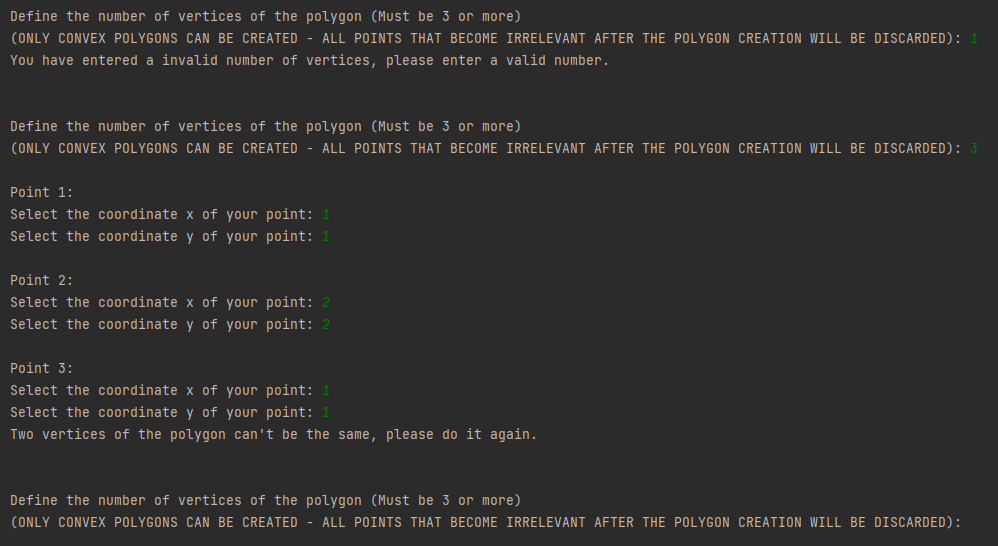
However, this can potentially generate confusions in how to connect them, leading to ambiguous types of polygons or even self-crossed polygons, as can be shown in the example bellow:

**Implementation:**

Example (Program.java):

****Execution sucsseful:

****Execution with error cases:

All the source code is available at is present moment in a separate annex, and at its last update in the repository.

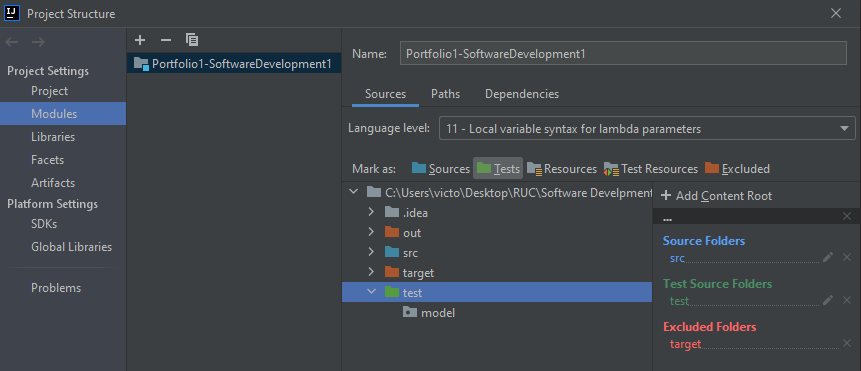
Plenty of functions were made and tested in or order to make the model work.

* At this present time (30/04/2022), only the functions “computeArea()” and “checkPointInside(Point p)” in the Shape class are still having some issues when working with polygons (but they work perfectly with circles).
* All the other functions are working properly and tested.
* The code is also commented explaining in more depth the implementation.

**Testing:**

All the tests were done using Junit 5 for all the non-abstract classes from the model package (“ClassTest”, “PointTest” and “PolygonTest”).

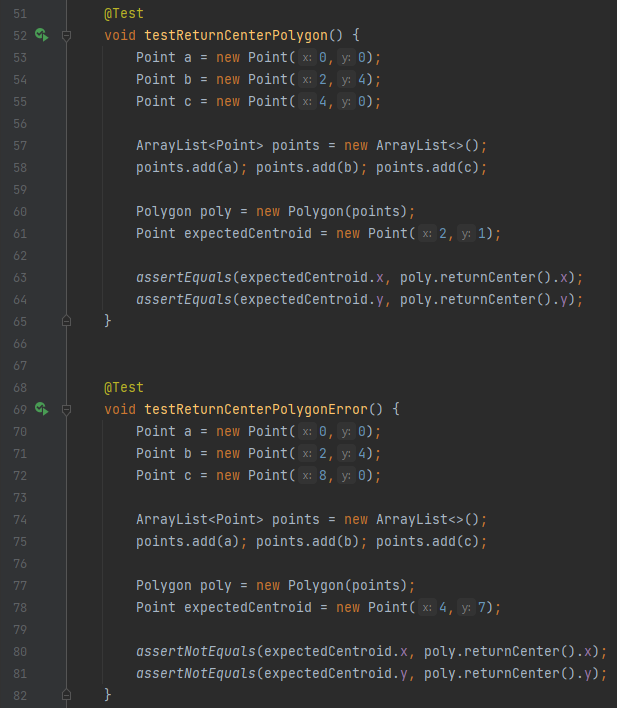
It was specified in the ‘Module Settings’ from InteliJ a folder called “test” to have all the unit tests required.



To test all the classes from the model, the “test” folder needed to have the same package inside of it.

All the tests followed the same principle of testing cases of success and error by importing assertions from the JUnit library.

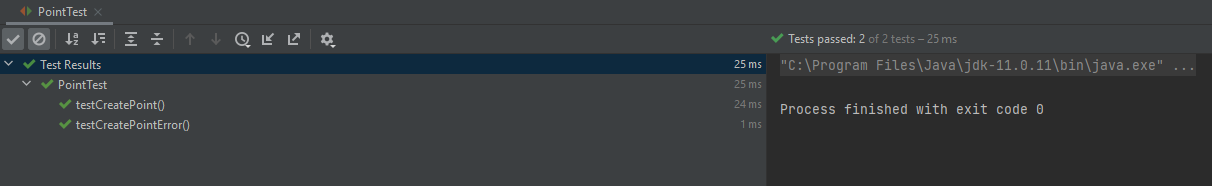
Example (PolygonTest.java):



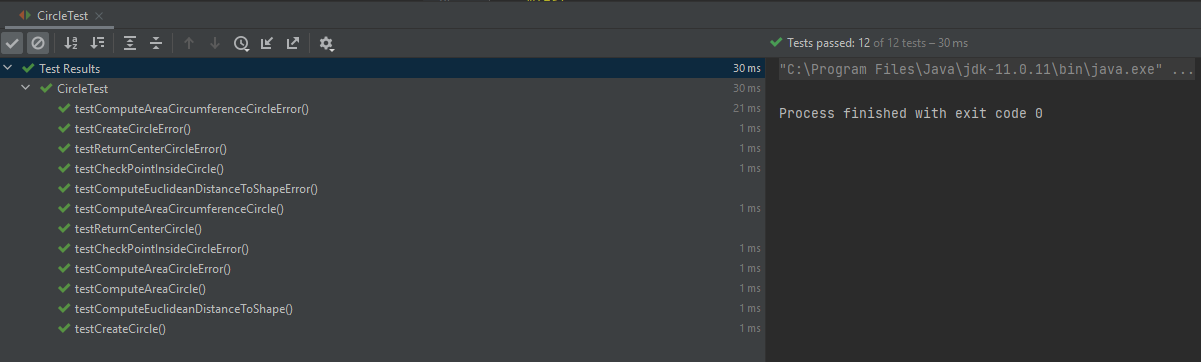
The tests were build this way because they minimize the chance of having false good results by actually exploring more possibilities of errors inside the model classes.

*Tests results:*

PointTest.java:



CircleTest.java:



PolygonTest.java:

