SENG2200 Assignment 2 Report

1. Progress tracking

Based on the feedback from Assignment 1, my first goal was to implement a Node class. All of the Node-based functionality was in the Polygon class prior to this. This was a simple 5-minute coding process – no design time was required and there were no errors to correct.

The PlanarShape class was next. I only spent a couple of minutes transferring and fixing some syntax errors in the compareTo() method based on more prior feedback, as most of the (now-derived) methods were implemented in the Polygon class. I also decided to make PI a protected attribute in this class, as it is used by a couple of derived classes.

Now the parent was done, I had to design the derived classes. Polygon was covered in Assignment 1 and Circle and SemiCircle had no new methods, only different implementations of the methods already in Polygon. Understanding the maths, designing and coding took about 30 minutes.

The trickiest part was implementing the iterator. Admittedly, my knowledge was a bit patchy so after revising, I spent about 4 hours researching, designing and coding the LinkedIterator class. Another 6 hours was spent re-designing and re-coding the main() method and LinkedList class to remove all the references to the current Node.

I implemented the Iterator in stages: first, I re-coded and debugged the output to exclusively use the iterator (about 2 hours), then the inserting functionality (*much* longer). There were a few bugs here caused by a persistent series of infinite loops. The program would search for the end of the list, only to skip over the sentinel node – things like that.

The final bug was a frustrating output bug – when presenting the lists, the iterator would continually skip over the second shape in the list (the next one after the sentinel node).

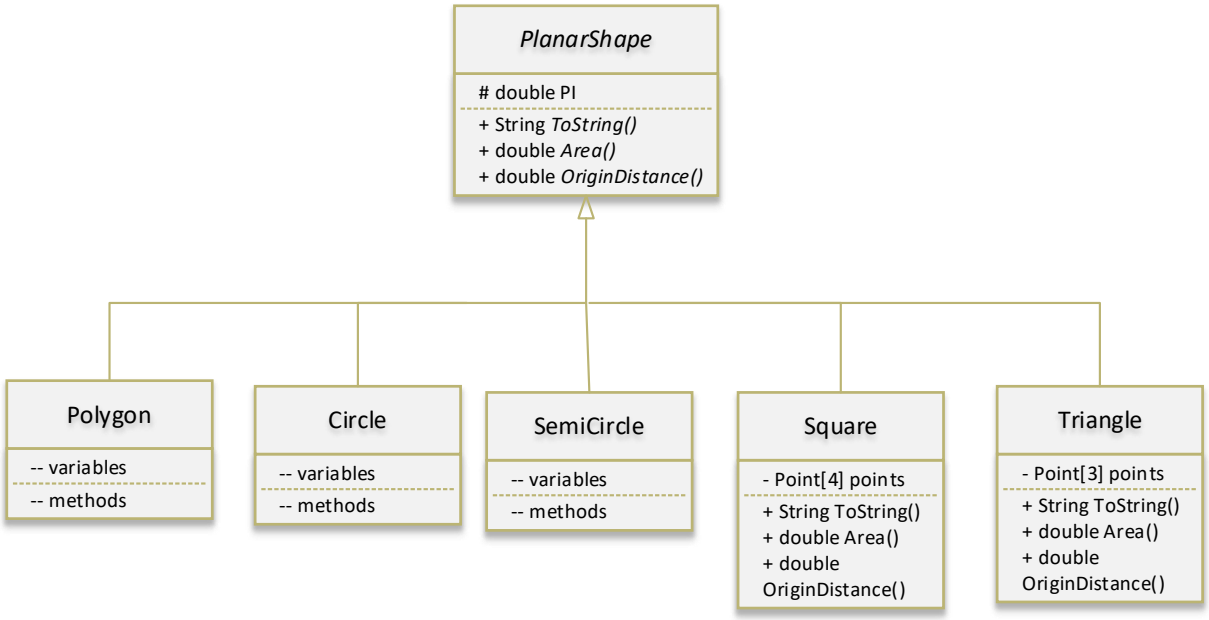
2. Error tracking

A greater proportion of errors came from the design when compared to Assignment 1. This is entirely because of my misunderstanding of how to implement a **custom** iterator.
About 30% of my errors came from design errors, with 70% coming from coding errors.

3. Extending the program to incorporate Triangles and Squares

First, the program needs to recognise a Triangle or Square from the input in the same way that it currently recognises Polygons ("P"), Circles ("C") and Semi-circles ("S"). Then we would need to create the derived classes Square and Triangle, both implementing PlanarShape. These classes would need the usual Area(), ToString() and OriginDistance() methods.

Like the Polygon class, these two classes both require an array of points. Unlike the Polygon class however, these arrays have a set length: 3 for Triangles and 4 for Squares. These new classes are displayed in the bottom-right of the following UML diagram:



4. Extending the program to incorporate Ellipses

Ellipses are not nearly as straight-forward as any of the other shapes that are covered in the scope of PA2. For example, the curvature of the ellipse is defined using two points of **eccentricity**, or *e* – the shape is a circle where *e* = 0, and a parabola where *e* = 1. An ellipse is any shape between these two points.

For the sake of modelling an ellipse, I would require an origin point (the middle of the ellipse), a major point (a point on the ellipse that is **furthest** from the origin, along the **semi-major axis**) and a minor point (a point on the ellipse that is **nearest** to the origin, along the **semi-minor axis**). These three points are the minimum points required to be able to calculate the area of an ellipse. The area is defined as **πab**, where **a** = major point – origin, and **b =** minor point – origin**.**

The origin distance would be more complicated to calculate. One possible way is to determine which of four possible points on the ellipse is closest to the origin: the 2 points along the **semi-major axis**, and the 2 points along the **semi-minor axis**. Luckily, it is easy to calculate all 4 points using the current attributes – the opposite point of each axis is twice the distance to the origin. This functionality may be incorporated into the OriginDistance() method.