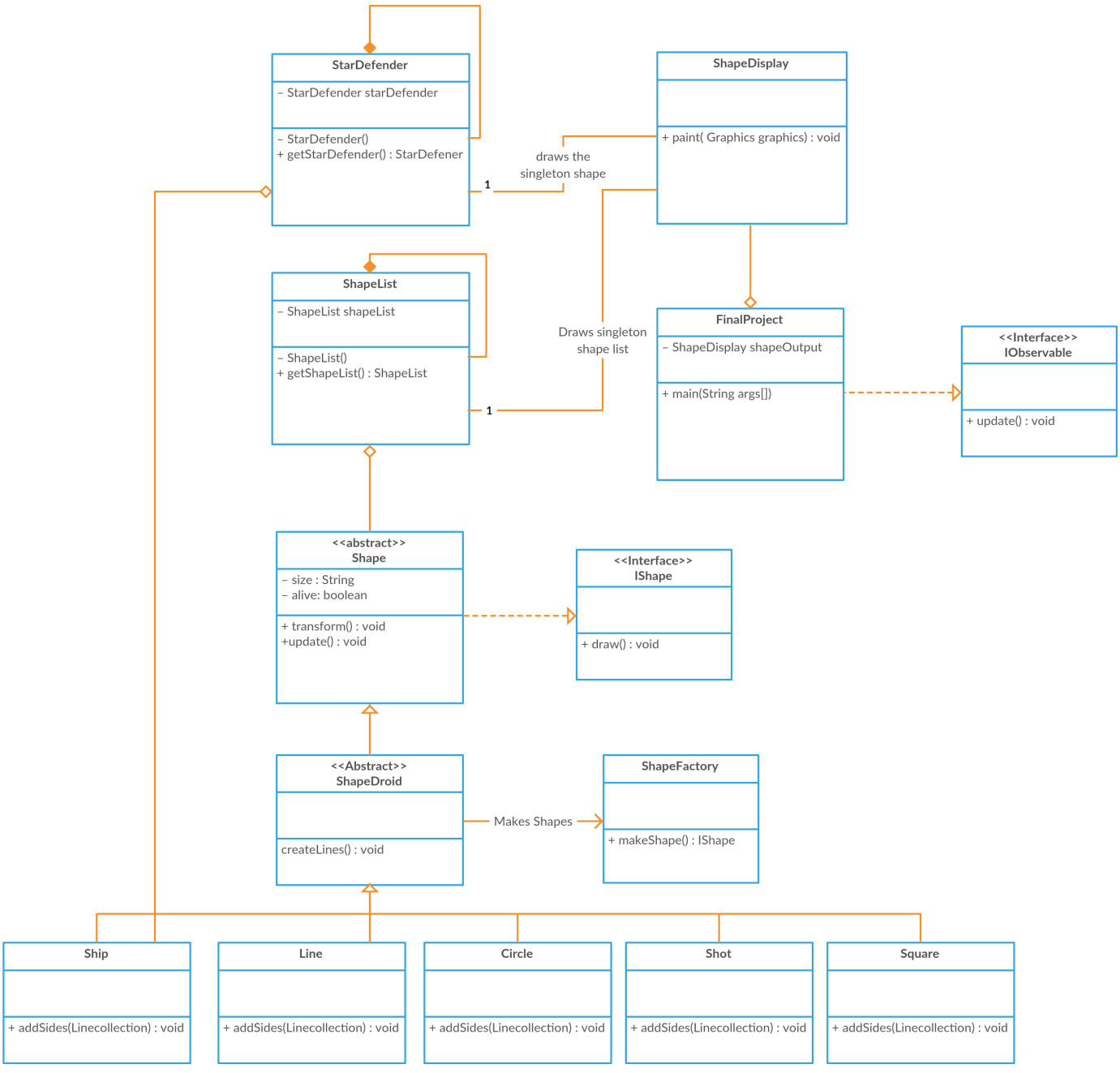
Tarek Nabulsi

SE 450 Final Project

Star Defender

(a.k.a. Asteroids)

UML Diagram


StarDefender
– StarDefender starDefender
– StarDefender()
+ getStarDefender() : StarDefener
1
draws the singleton shape
ShapeDisplay
+ paint(Graphics graphics) : void
ShapeList
– ShapeList shapeList
– ShapeList()
+ getShapeList() : ShapeList
1
Draws singleton shape list
FinalProject
– ShapeDisplay shapeOutput
+ main(String args[])
<<Interface>>
IObservable
+ update() : void
<<abstract>>
Shape
– size : String
– alive: boolean
+ transform() : void
+update() : void
<<Interface>>
IShape
+ draw() : void
<<Abstract>>
ShapeDroid
createLines() : void
Makes Shapes
ShapeFactory
+ makeShape() : IShape
Ship
+ addSides(Linecollection) : void
Line
+ addSides(Linecollection) : void
Circle
+ addSides(Linecollection) : void
Shot
+ addSides(Linecollection) : void
Square
+ addSides(Linecollection) : void

Patterns

In this program of asteroids I made good use of the singleton, factory, and strategy patterns. I used these three in particular for a few reasons. First of all, when initially tackling this project, I broke down all the shapes that needed to be drawn into three different categories: enemy shapes, player ship, and shots fired from the ship. Since they all act slightly differently with each other, I decided a singleton pattern that stores each individual type would be more effective for keeping code concise, maintainable and easy to implement. This also made it particularly straightforward when checking for collisions between the shapes. Since I never needed to check for collisions between shapes within the same singleton class, all I had to do was check between shapes of one singleton class with that of another, specifically between the player ship and enemy shapes, and between the shots and the enemy shapes. Overall this pattern was useful because it provided global access to the objects as well as guaranteeing that the object would only be instantiated once. This was especially important for the player ship since there can only be one player ship in order to play the game.

The factory pattern was also very useful for the project. The shape factory class helped create the shapes from the JSON file since it could automatically determine which type of shape to make based on the string input, and it could process it all automatically. Additionally, I created another factory for the shots. This was useful for automatically creating a specific type of shot object and adding it to the shot singleton every time the fire button was pressed. With this factory, all the shot singleton class had to do was call the "make shot" method from the factory and the shot would be made.

Lastly, the strategy pattern was very useful for determining how a shape would behave with other shapes and the borders. This pattern allowed me to conveniently copy and paste a strategy object as a parameter for a shape object rather than having to rewrite the methods in each class. I wanted the circle shapes to rebound off the border and the squares to pass through. However I also wanted to have the ship pass through as well, so the strategy eliminated having that repetition in my code. I also created a new strategy called "end strategy" for the shot objects. I made this strategy because I didn’t want the shots to rebound or pass through the border. Rather, I just wanted the shots to disappear once they hit the border, which is what the strategy pattern specifies. Overall, the strategy is very useful for changing the way the objects behave in runtime. It is also very interchangeable, easy to maintain, and eliminates repetition between class behavior.

_ _ _

Discussion

The main problem I sought to address throughout the programming of this assignment was to eliminate repetition of code as much as possible. I knew this would allow me to create a more complete and aesthetically pleasing game this way. The more efficient the code, the easier it was recycle and implement new features. The singleton and strategy pattern helped me easily create the shield feature, which is activated by pressing the 'V' key. Once pressed, a five second timer is started, and the shield is deleted once that timer ends. Other buttons I included were the start button – '1' – pause button –'T' – and quit button –'Q.' The first two of which were easily done by implementing the observer pattern to simply start and stop observing of the shapes. Another button I used to help with the developing of the game was the hyperspace button – 'B.' Pressing this simply replaces the player ship with new player ship in a random location within the boundaries of the window. It still creates a new player ship even after the initial ship is destroyed by the collision with another shape. This allowed me to keep the program running rather than having to restart it.

The main issue I had with the assignment was detecting the collisions. The detection with the shot is very inconsistent and may require a few shots to be fired at a shape in order for the collision to be detected. This is most likely due to the shot's small size since collision with the ship never fails to be detected. Another issue was keeping track of memory and space. Since the shapes divide into two more shapes after being hit, and three lines are made as debris for each time as well, I had to make sure each of these objects and their line collections were properly deleted once they were no longer being drawn to the GUI. My program would crash during development for this reason.

Aspects that made this assignment easier included the configuration file. This allowed me to easily test different numbers in the settings to find what worked optimally rather than fishing through to code to find where each number needed to be changed. This kind of option also resembles what might occur when a user changes the customizations in the settings menu of an application or game. The program takes the values from that menu or interface, parses the data if needed, and stores each value in their relevant locations to allow the application to run. The one setting I chose not to use was the shot lifetime, since I wanted the shot to end exactly when it hits the edge of the window and not bounce back or pass through. I also limited the player to only firing three shots at a time to prevent too many shots from being fired at once. Lastly, another feature I included was putting a limit on the ship's speed in the method, foo velocity. This prevents the ship from flying out of control if the thrust key is kept held down for too long. Once shapes move too fast, only a few of its locations get drawn to the GUI and could serve as a potential problem for detecting line collisions.

This project definitely helped me understand solving real world problems that may arise when programming an application. It is one thing to add a feature, but it is often more important to make the code simpler, more concise, and easy to maintain, allowing patterns to take care of most of the work for you. This way when bugs arise, or if a feature needs to be changed, the chances of breaking something else become more slim. Additionally, the speed and efficiency of the code's execution make a great difference in the software product as well. Overall much was learned from taking on this assignment.