Rocket Odyssey

Analysis and Design Document

Student: Tofan Emil – Mihai

**Group: 30432**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 03/04/2021 | 1.0 | Initial Revision | Tofan Emil-Mihai |
| 23/04/2021 | 2.0 | Added Chapter III : Elaboration : Dynamic Behavior and Design Model | Tofan Emil - Mihai |
| 26/04/2021 | 3.0 | Added Chapter III : 1.2 Class Design : The Strategy Pattern | Tofan Emil - Mihai |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

# Project Specification

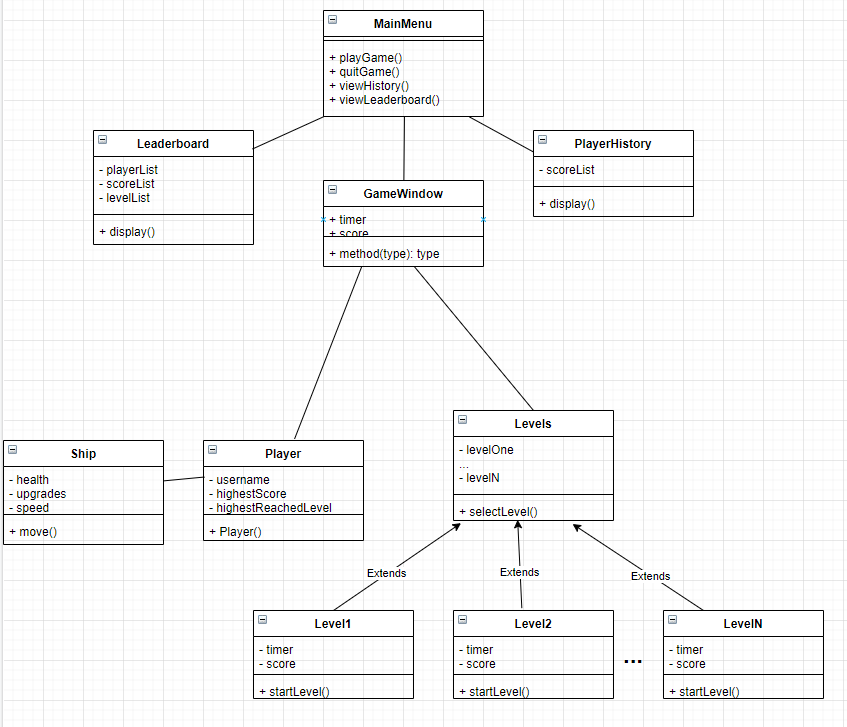
This project will resemble an old arcade game. The user will be able to guide a rocket through rains of asteroids and enemy fire to victory. For every second of gameplay, a score will increase and will be displayed in the upper left corner of the game, the user’s goal being to obtain as high of a score as possible. As time progresses and the score increases, the user will advance from one level to another, being presented with a higher level of difficulty. The user will be able at anytime to pause the game and take as much time as he or she needs, being able to start again from the moment they have paused at. Besides the score, an HUD will display the rocket’s condition in terms of health. The health will decrease everytime the rocket takes damage and will end the game if it reached zero. The user will also to pause the game and enter a shop, from which he or she can purchase certain upgrades for their rocket which will help overcome obstacles with more ease.

Besides the effective gameplay, a user can also check his or her score history to see if they have improved at the game or not. Each user of the application will be required to enter a username and based on this username, the history will be created. For every different user, a rankings window will be available so they can compare their scores with those of other users and try to reach the number one spot in the leaderboards.

# Elaboration – Iteration 1.1

# Domain Model

The domain model will incorporate the menus described above. For the types of objects, I’m thinking of having a player, a ship and levels. I’m inclined to create a level object instead of just using it as a parameter for the player object to incorporate more logic behind each level. The conceptual diagram is shown below, all items presented are subject to change.

**

# Architectural Design

## Conceptual Architecture

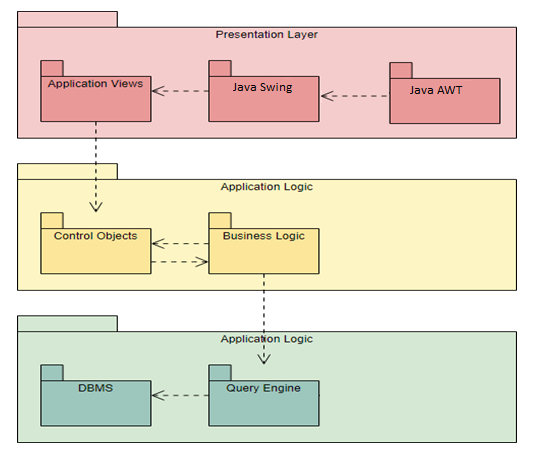
From what I know, there isn’t really any standard pattern for game development because every game is different. There are basic techniques that work in all games, usually found as a library, but nothing as strict as a design pattern. For a conceptual description of the architecture, the components I’m thinking of are listed below :

* A graphics engine which will help draw the graphics for the computer to display.
* The audio/sound engine will be dealing with the sound effects embedded in the game.
* Another component of my architecture would be the I/O components in order to allow the mouse, keyboard, possibly joystick and other peripherals make the game more interactive

When talking about the connection with the database, I will be using the layered architecture for all the objects that will be stored in the database. Those objects that will be displayed in the match history and in the leaderboards like the usernames, the highest scores, the history of highest scores with the date and the maximum reached level.

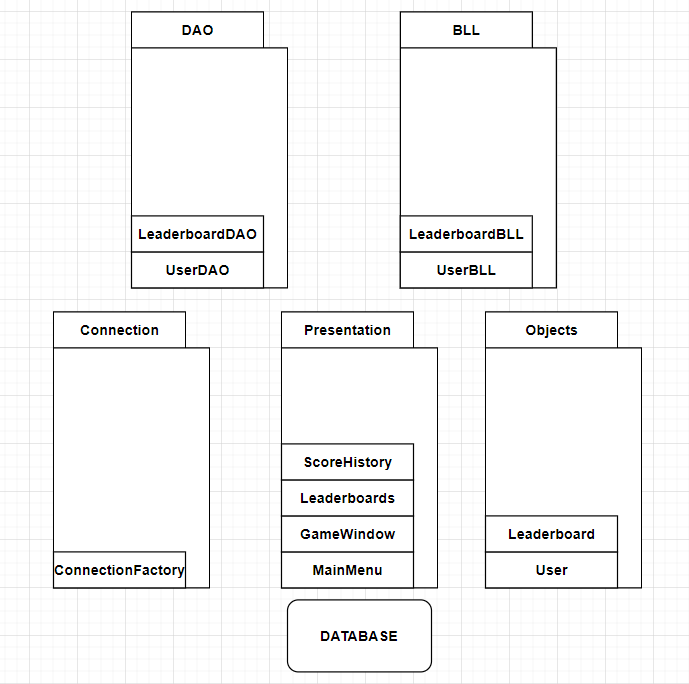
## Package Design

**Package Diagram**

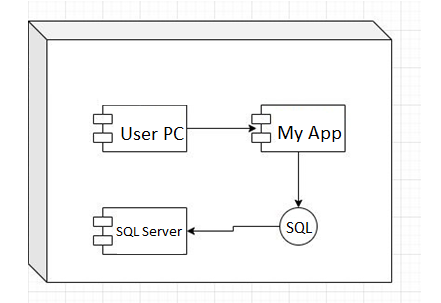
**

## Component and Deployment Diagrams

**Component Diagram**



**Deployment Diagram**

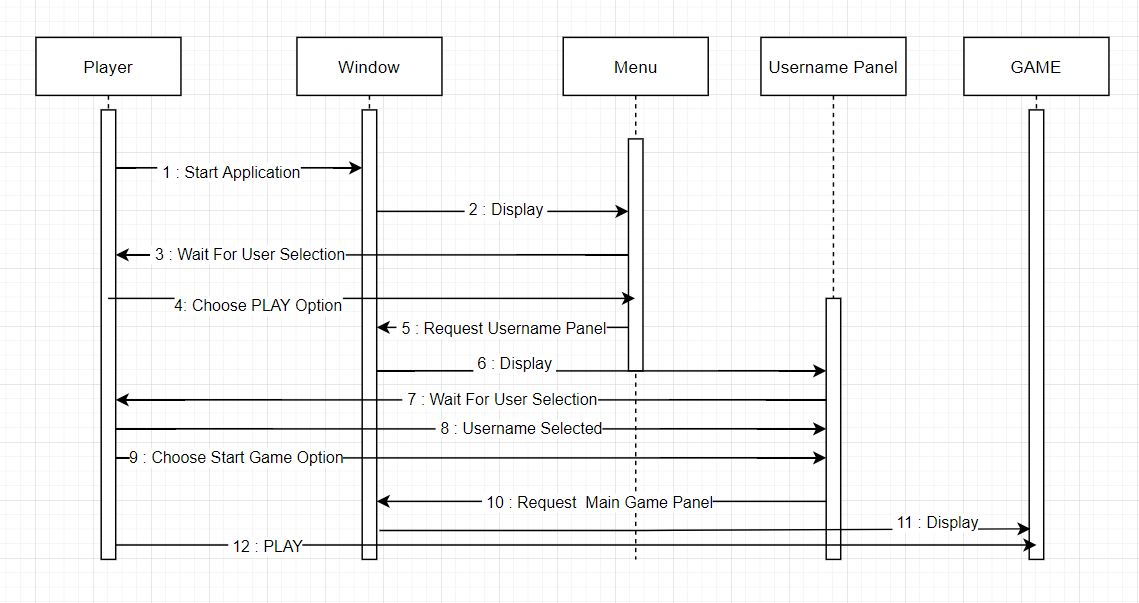
****

# Elaboration – Iteration 1.2

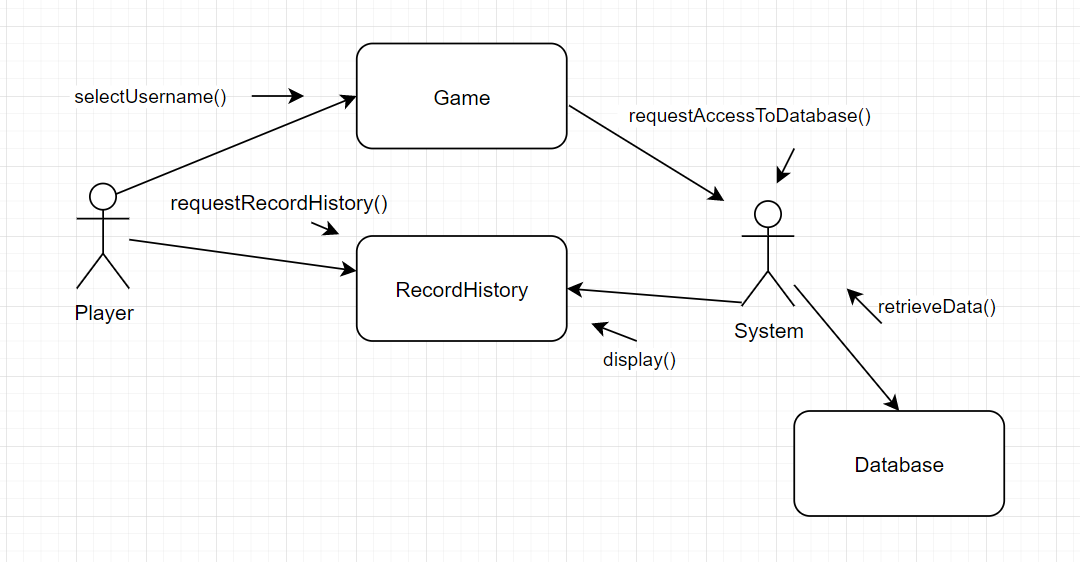
# Design Model

## Dynamic Behavior

**SEQUENCE DIAGRAM**



**COMMUNICATION DIAGRAM**



## Class Design

Regarding class design and the implementation of design patterns withing my application, I will start presenting each design pattern used, its role, the reason I chose to use it and motivate my reasoning.

**Factory Method Design Pattern**

The first design pattern implemented in my application is the Factory Method Design Pattern. It is a creational design pattern, widely used for the creation of objects without specifying the exact class to create. It was really helpful in my application, since I needed many objects to be created over the different levels of the game, and it offered me a way to encapsulate the creation of those objects. It helped me centralize class selection code, got rid of scattered object creation within my application and provided a way for the classes to be chosen at runtime.

For the implementation of this design pattern, I had implemented an abstract class called GameObject which would define the basic methods all my game objects would inherit. In order for the implementation to work, I had to come up with a way to distinguish between my objects and for this, I had created an enumeration called ID, which would store ID’s for any subclass that extended my abstract class. Even more, the same subclass can have different ID’s. The same object would be created, but certain characteristics would differ (in my case, the image of the object)

After this, the implementation of the GameObjectFactory class came. In this class, the constructor for the factory would take as parameters the same parameters a normal game object would take, those being :

x (int) = the position of my object in regard to the x axis

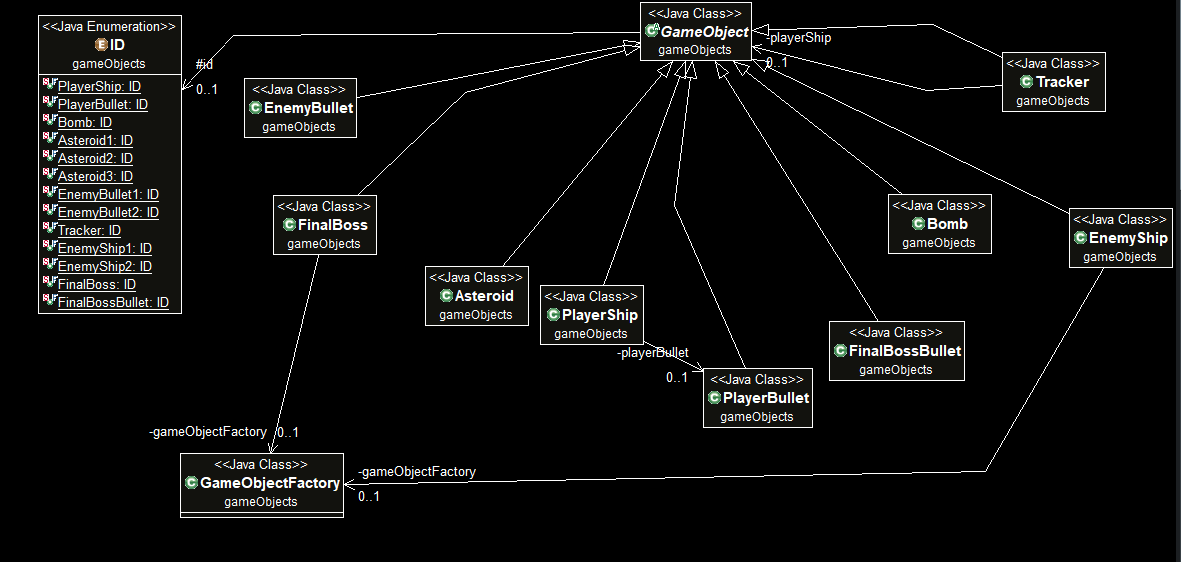
y (int) = the position of my object in regard to the y axis

id (ID) = the unique identifier for any object

handler (Handler) = the game component which keeps track of all the objects present in the game, adds new ones, deletes old ones and adds them by rendering them

After the creation of the factory, the design pattern had been implemented successfully, and its advantages were clear, especially when many objects needed to be created, for example in my spawner class, which deals mainly with the object creation logic for each level of the game.

The UML diagram describing the classes involved is displayed below.



**Observer Design Pattern**

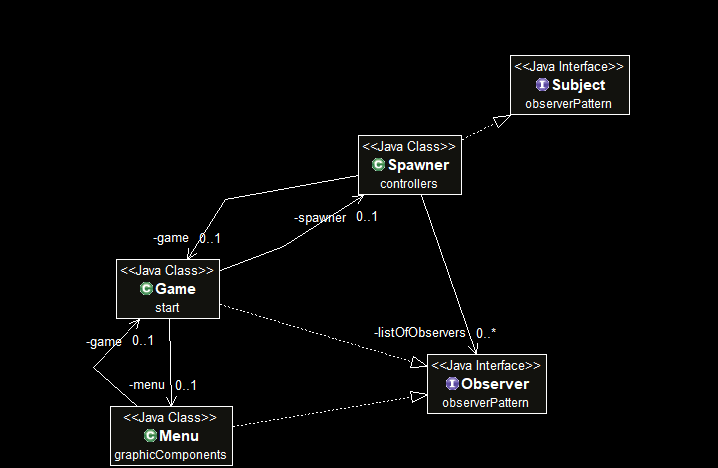
The second design pattern I implemented was the observer design pattern. It represents a behavioral design patterns typically used when we want a number of observers to be notified of a certain event. I chose to implement this pattern starting from the idea that I want my game to be as interactive with the player as it can be, and notifications about certain events is what came to my mind, the observer design pattern being the best for the job.

For the implementation of this design pattern, two interfaces were defined :

The Subject interface which contains two methods : registerObserver() which adds an Observer to the observers list and the notifyObservers() method which send a notification to every Observer previously added in that list. This interface was implemented by the spawner class, which mainly holds the logic for each level of the game. My idea was that, when the player would hit level ten, the final level of the game, the boss would normally appear. Being the most powerful NPC from my game, I wanted my game to let the user know that the boss is approaching. The spawner class, holding the logic for the boss’ appearance, will call the notify observers pattern then.

The Observer Interface is the second interface defined and it is the one which contains one method called update and was implemented by the Game class and the Menu Class. The method update is implemented differently for those two classes. Upon being notified, the game class is the one which will display the notification to the player by calling another class called BossNotification. The menu class is the one in which the player ship is defined, and because of the boss notification and the game window overlaying, the game window would stop receiving input from the player keyboard and the ship would keep moving. Thus, the Menu class was added as an observer and, upon being notified, it would stop the player ship.

The UML diagram describing the classes involved is displayed below.

****

**Strategy Design Pattern**

The third design pattern implemented for my application is the strategy design pattern. It represents a behavioral design pattern typically used when there is a need for an algorithm to be selected on-the-fly at runtime. It was really useful in my implementation because I needed a way to change the behavior of one of my NPC’s and the strategy pattern provided just the right functionality.

The NPC I am talking about is the Tracker. Different from the other NPC’s, which have a rather fixed way of moving withing my application, the tracker changes its direction at runtime based on the distance between itself and the player’s ship. Using the algorithm for calculating the Euclidian distance, the tracker will modify its position on the map and ,,track” the player’s ship with the purpose of hitting it and inflicting damage. In order to add another functionality to the tracker’s mode of operation, I implemented the strategy design pattern as follows :

I created an interface called Strategy containing only one void method, createTrackerStrategy which would take as a parameter a tracker.

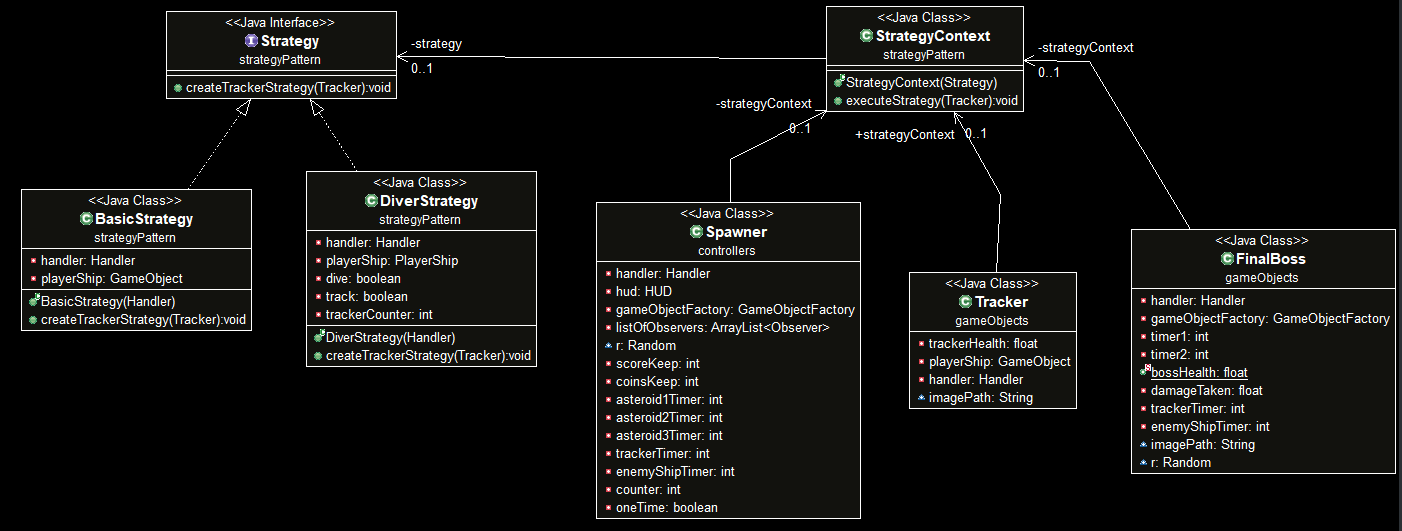
Implementing this interface, there are two classes called BasicStrategy and DiverStrategy. They each override the createTrackerStategy(), each with their own implementation for the way in which the tracker should behave around the player ship.

The Basic Strategy would make the tracker NPC continuously track the player at a low speed. The Diver Strategy would make the tracker more tricky to play against. The tracker would slowly track the player around givind the impression that it is just a simple tracker. After a certain amount of time, the tracker would suddenly distance itself from the player for a few seconds, only to dive right into the player at fast speeds afterwards, making it very difficult to play against.

The next step was to implement the context for which these strategies would be used. For this I implemented a class, called Strategy Context, which would take as a parameter in its constructor a Strategy, and have one method called executeStrategy() which would call the specific createTrackerStrategy() method, corresponding with the Strategy object with which it was created.

In order to pass the strategy which will be used to a Tracker NPC, the tracker class has a Strategy Context attribute and a setter for it. Inside all the classes which require the creation of a Tracker (The Spawner Class, which spawns trackers at different levels and the FinalBoss class which randomly spawns trackers while fighting the player), a context variable would be created using the desired strategy together with a tracker object. The tracker object’s context would then be set with the previously mentioned context and thus, the tracker would obtain the new strategy of following the player ship at runtime.

The UML diagram describing the classes involved is displayed below.



# Data Model

For the data model, I will be splitting the data into two categories :

- The data used inside the database

- The game data used for storing the different types of objects present in our games

For this system’s implementation, two data models regarding the database have been used and they are the leaderboard and the recordHistory

The leadearboard object was created in order to keep track of the best games players have ever had. In the game window, a user can access the leaderboard and he will see the top ten scores achieved by all users over all games played.

The recordHistory object was created in order to keep track of all the games across all players. In the game window, after a user has selected an username, he or she will be able to access their private record history where they will see the history of their last ten played games.

Each of these data models have their separate table in the database and they are described below.

The Leadearboard contains four colums :

- idLeadearboard : the unique identifier for any leadearboard object

- username : the string representing the username used by the player in the game

- highestScore : an integer storing the highest score obtained by that particular player over all the games played

- highestLevel : an integer storing the highest level at which the player has arrived, in the game for which he obtained the highest score

The recordHistory table contains four columns :

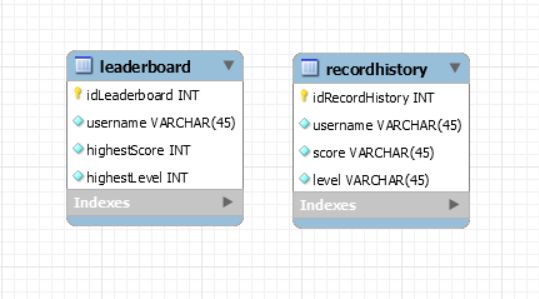
- idRecordHistory : the unique identifier for any recordHistory object

- username : the string representing the username used by the player in the game

- score : an integer representing the score achieved by the user for that particular game

- level : an integer representing the level at which the player has arrived, in the game for which he obtained that score

The database diagram for this application can be observed below



Moving on to the data used inside the game, each entity which is displayed on the screen at a moment in time represents an object. These game objects, as I called them, represent subclasses of an abstract class called GameObject. In total, there are nine such objects and their names are :

- PlayerShip

- PlayerBullet

- Asteroid

- Bomb

- EnemyBullet

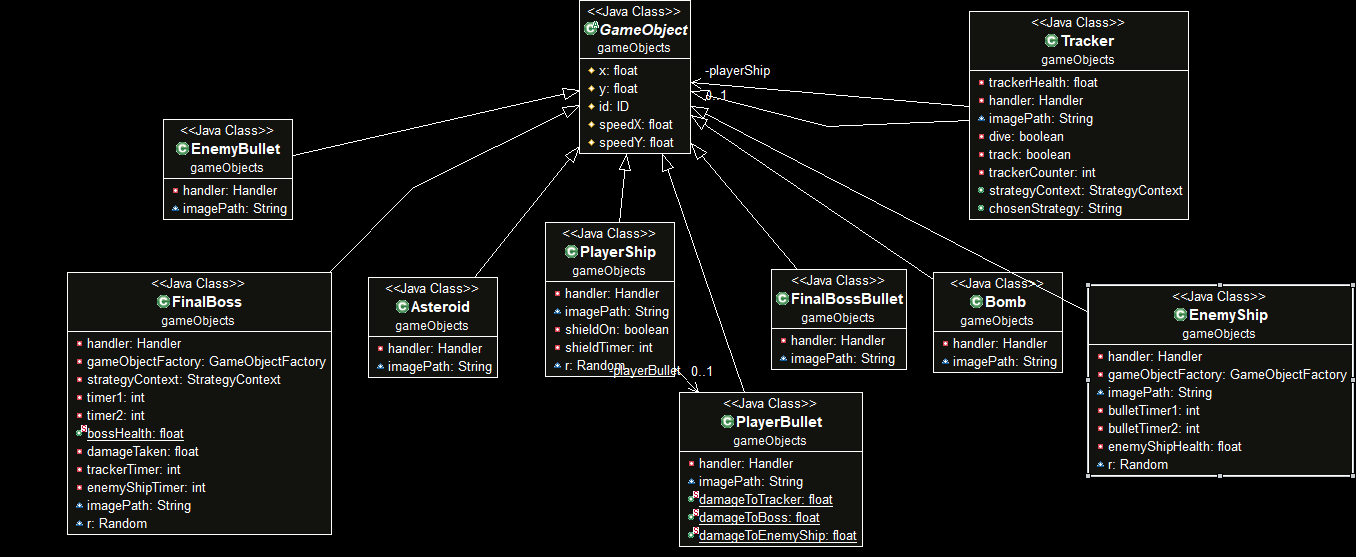
- EnemyShip

- Tracker

- FinalBoss

- FinalBossBullet

The UML diagram for these objects is displayed below :



More about the creation of these objects and their relationships can be seen above, where the Factory Pattern and its implementation withing my application was described.

# Unit Testing

*[Present the used testing methods and the associated test case scenarios.]*

# Elaboration – Iteration 2

# Architectural Design Refinement

*[Refine the architectural design: conceptual architecture, package design (consider package design principles), component and deployment diagrams. Motivate the changes that have been made.]*

# Design Model Refinement

## *[Refine the UML class diagram by applying class design principles and GRASP; motivate your choices. Deliver the updated class diagrams.]*

# Construction and Transition

# System Testing

*[Describe how you applied integration testing and present the associated test case scenarios.]*

# Future improvements

*[Present future improvements for the system]*

# Bibliography