Software Design Document

for

WizardingWorldGame

Team: Team Three

Project: WizardingWorldGame

Team Members:
Priyanka Ghosh Dastidar

Last Updated: [4/4/2023 8:00:26 PM]

Table of Contents

Document Revision History

Revision Number	Revision Date	Description	Rationale
1.0	April 4th, 2023	Deliverable 2 document	Design document includes. • An introduction to the Project • An overview of the software architecture • Description of the sub-system services • UML diagram for each section of the services

List of Figures

1. Introduction

WizardingWorldGame is based out of the Harry Potter, a young wizard and his archenemies who are all centered at Hogwarts School of Witchcraft and Wizardry. The focus of the gameplay is to allow the main character(Harry Potter) to dodge endless waves of different projectiles coming in from its archenemies. In this game, the player will control the main character which moves around in the screen, in eight directions. The player will shoot at an array of enemies(Regular enemy [Basilisk, Dementor], mid-boss [Professor Umbridge]) until reaching the boss (main archenemy, here it will be Voldemort). The boss will be more powerful and have more elaborated and complicated attacks. Throughout the game, the player can achieve power-ups which will give them privileges, to protect themselves from enemy attacks.

For a player to win the game, the main character(harry potter) must survive all the stages of the game, after battling with an array of archenemies. For this Project, the end goal will be to design and implement a stand-alone desktop application. To implement this game, I have tried to use a clean software architecture, to ensure that the code structure is modular, extendable, and efficient.

1.1 Architectural Design Goals

Here, I have tried to implement a system that comprises of modular and loosely coupled sub-systems. The design goals discussed below, aims to ensure that the implemented system is free from any defects and contains all the required attributes to make sure runtime, system design and user experience are streamlined, efficient and proper.

1.1.1 Design Quality 1

The primary objective that the system focuses on is reusability of features. To make a codebase efficient, it should not have redundant code, the code should be organized in modules to make it easier to maintain and extend. In this codebase, I have made use of Factory method pattern, for laser movement and strategy and for different enemy types. This pattern allows us to create object, despite specifying their concrete classes. Implementing this, we can create a flexible and modular input system which can be modified or extended. It is quicker, simpler, and less expensive to design and produce a new product when designs are reused. This is since new components are much less reliable and tested than existing components. Because both development and validation time may be shortened, reuse offers the opportunity to launch a system or application as soon as possible. When delivery speed is not given a higher priority than overall development costs, it is very beneficial.

1.1.2 Design Quality 2

The next factor that I have looked for is scalability of the application. In recent times, most games they roll out new features to engage users to the game. So, to adapt to that setting, in this game, there is no hard coded settings in this game engine, all the configurations are imported from JSON files. The game engine reads from the JSON and likewise the character and their features are rolled over in display. This in general makes the game more scalable and easier to modify, as we need not do any code change to deploy new features in the game, rather we can make changes in the JSON to replicate such behavior. In my system architecture, I have used MVC (Model, View, and Controller) framework to separate out the subsystems in the program. This would in turn, reduce coupling between components. Core programming operations can be compartmentalized into their own, orderly boxes using MVC.

2. Software Architecture

2.1 Overview

The UML sequence diagrams are shown below representing the three use cases for this game, which comprises of Enemy spawns, and Player wins Game.

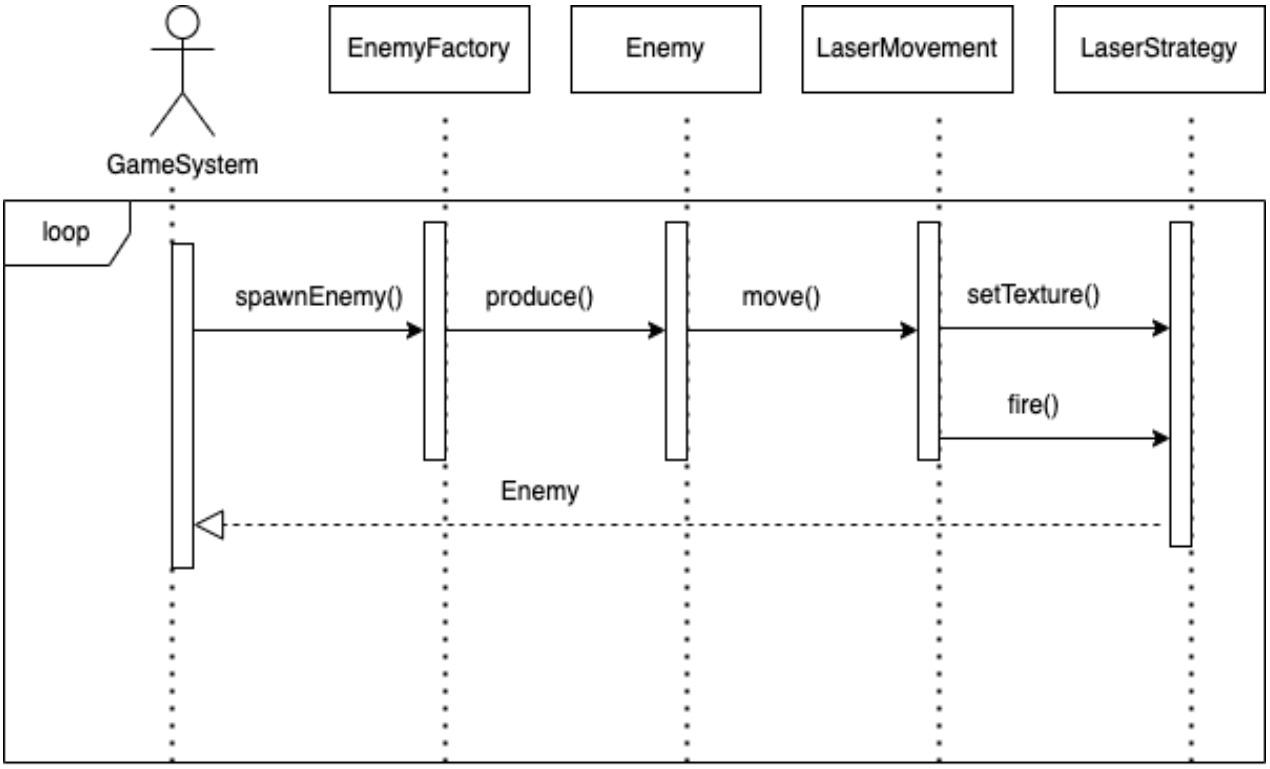

Figure 1-1. Enemy Spawns – Sequence Diagram

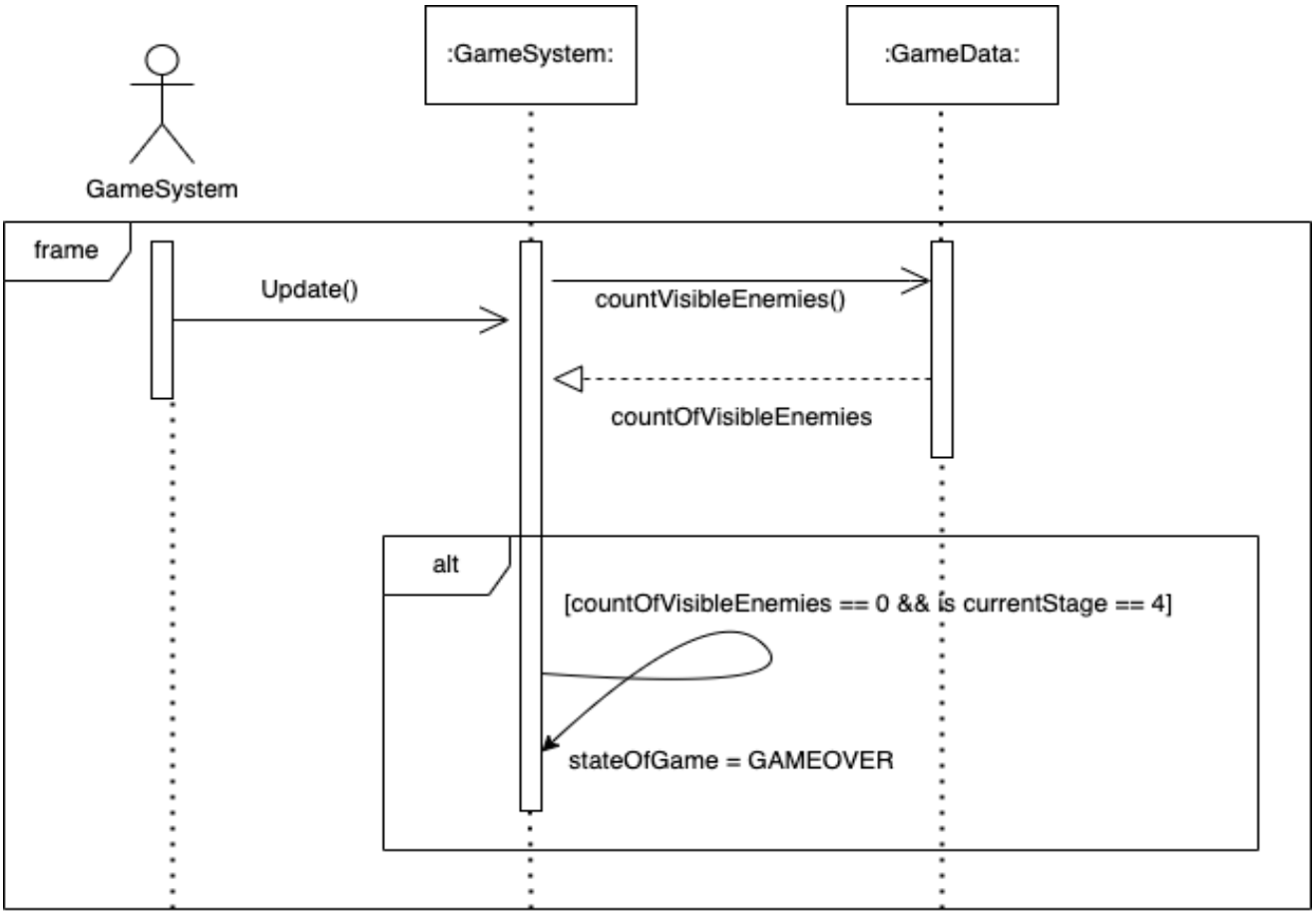

Figure 1-2. Player wins Game – Sequence Diagram

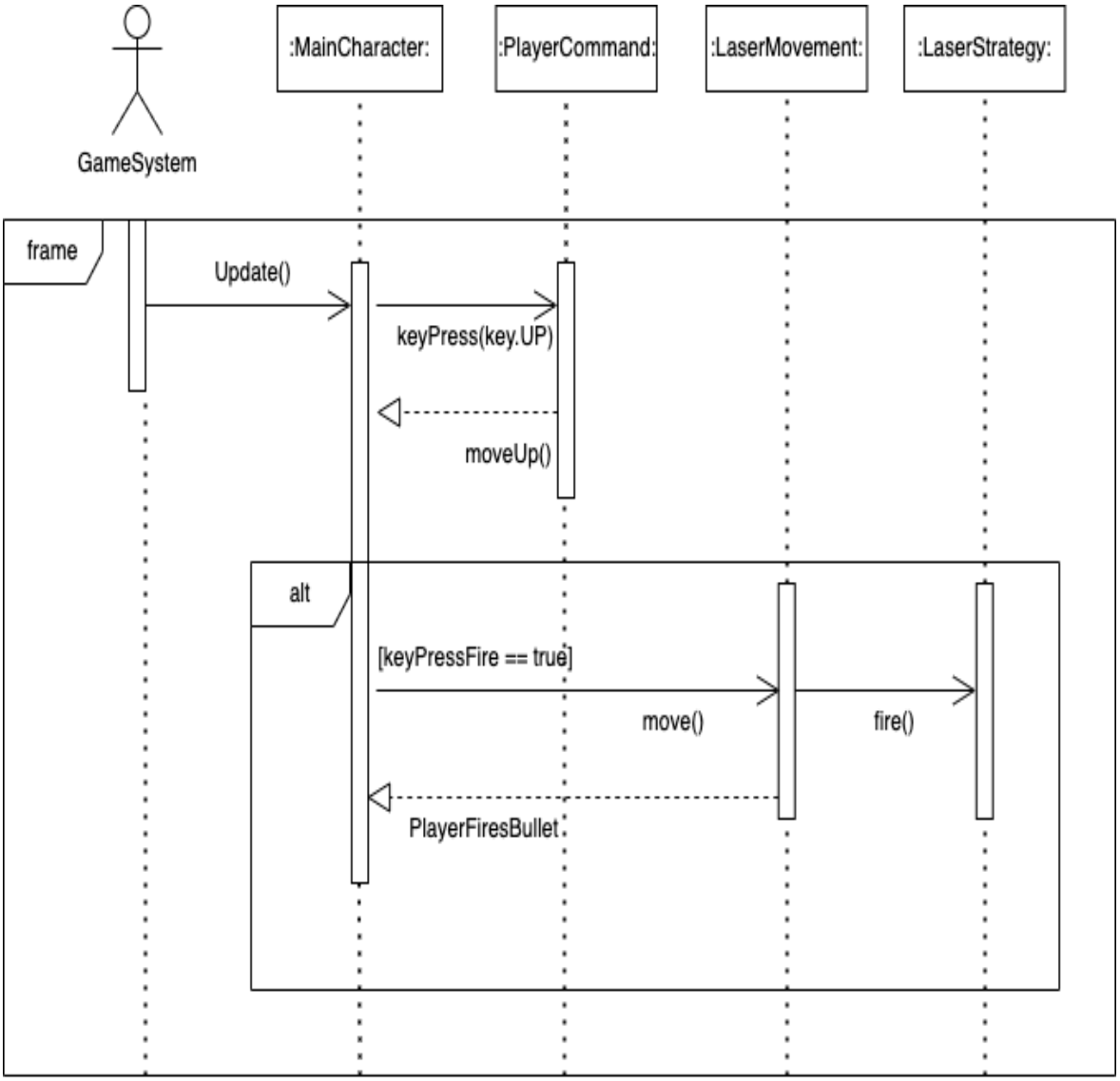

Figure 1-3. Player Shoots Bullet – Sequence Diagram

This System design follows MVC (Model View Controller) architecture pattern. MVC framework to separate out the subsystems in the program. This would in turn, reduce coupling between components. Core programming operations can be compartmentalized into their own, orderly boxes using MVC. In this application these components are represented by: Game Screens(View), Game Objects, Factories(Model) and Game Controllers including Laser Strategy, Game system, Player Commands and JSON Configuration Reader(Controllers). Modularizing these components, allow us to better organize and maintain code efficiently.

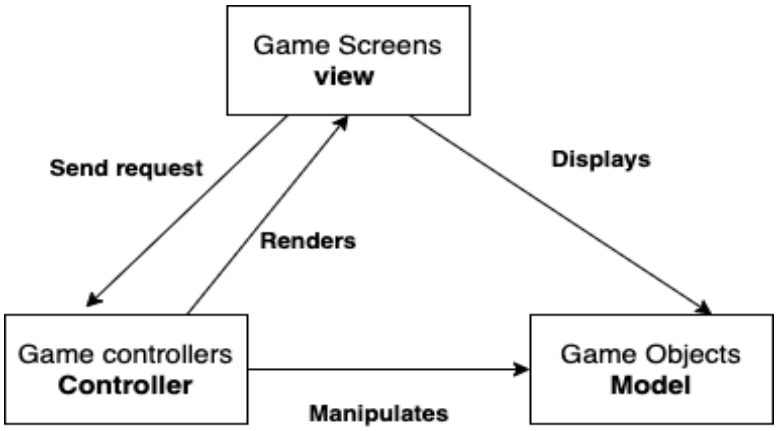

Figure 2-1. MVC Architecture – Component Diagram

2.2 Subsystem Decomposition

To utilize MVC architectural pattern, I have used Game Screens(View), Game Data (Model) and Game Controllers (Controllers). This in general will allow to render, store our data, and manipulate it in a more modular, cohesive, and scalable way.

2.2.1 Game Object Subsystem

Game object subsystem holds all the data needed by Game Screen and Game controller. It basically oversees organizing and storing all the data needed by the application for modelling purpose. In Game Object subsystem, data includes all Game Objects needed for modelling the game including Enemy, The Player, Laser movement and enemy factory and Game Controllable. In addition to the mentioned classes, it includes their creational and structural design pattern. This module exposes operation which corresponds to fetch and query data parallelly abstracting operations as much as possible. One of querying data includes an operation which involves in determining whether two game objects collided

with each other. In addition to that this module also contains data-related logic that the user works with and business logic data which the model uses. Game Object subsystem exists independently and has no knowledge of other two subsystems and works on operations asked by those two subsystems.

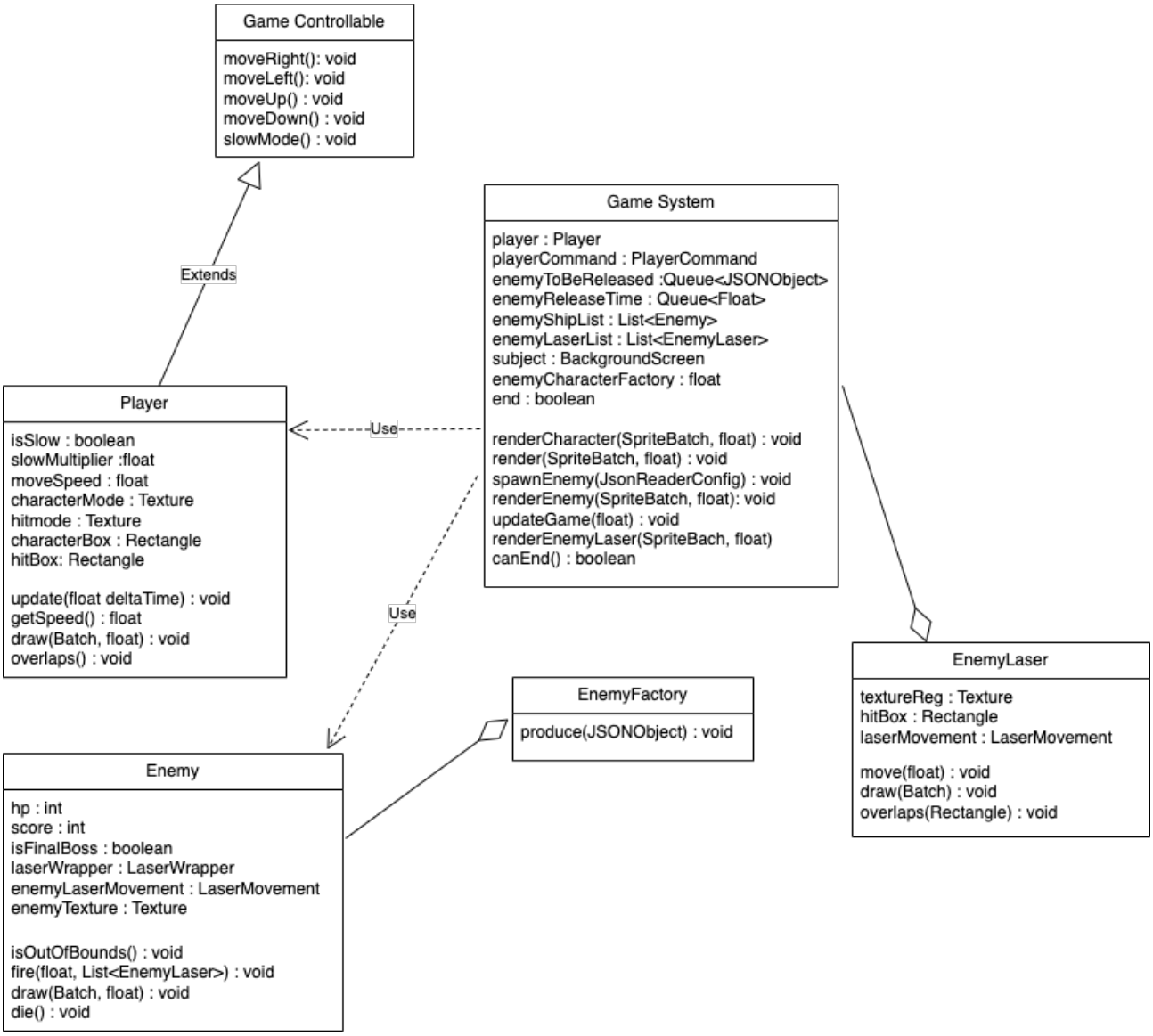

Figure 2-2. Game Object Subsystem

2.2.2 Game Screen Subsystem

This subsystem serves as a user interface and whatever is displayed to the user is via this module. It retrieves data from Game Data subsystem processing Game Object subsystem model and displays the content to the user and user is also able to modify the data. This part corresponds to the view module of MVC framework.

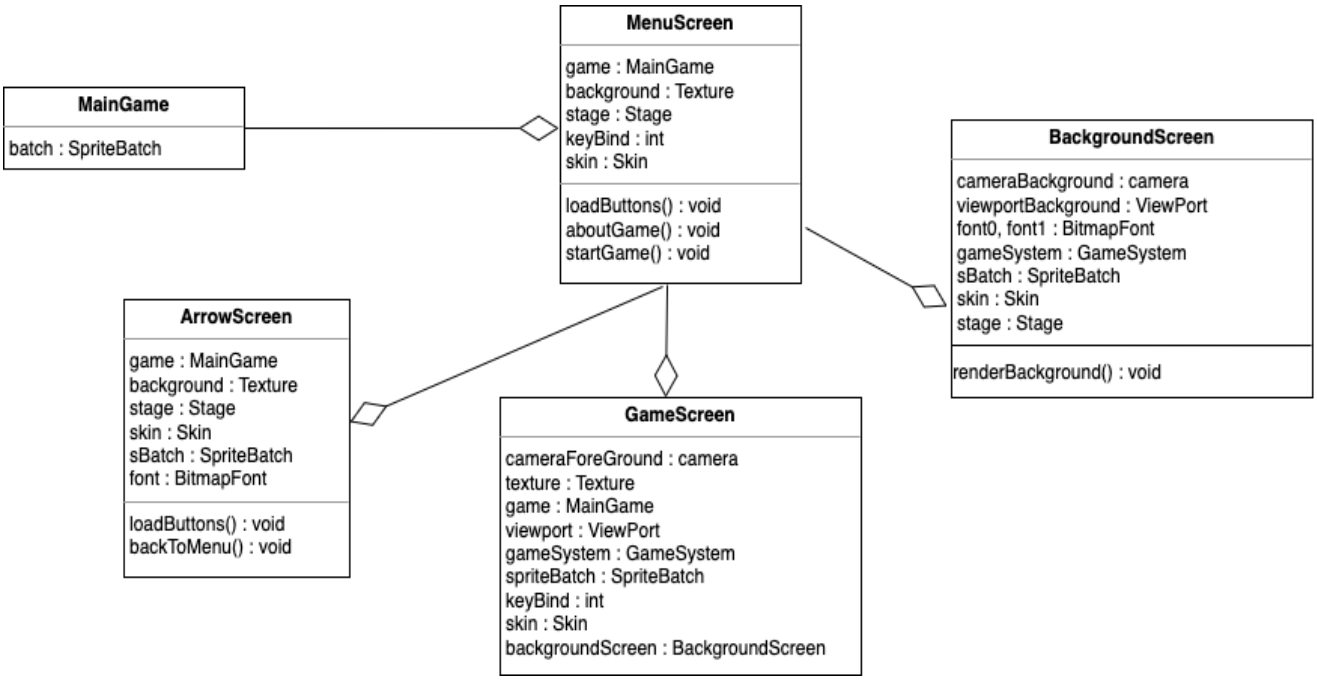

Figure 2-3. Game Screen Subsystem

2.2.3 Game Controller Subsystem

The game controller subsystem handles every user request. The controller module binds Game screen subsystem and Game Object subsystem. This works on the game data to modify and update as per the user request and then to reflect the changes through Game Screen subsystem. So, in general it serves as an interface to process logic and user requests and interact with Game Screen to render the display/output.

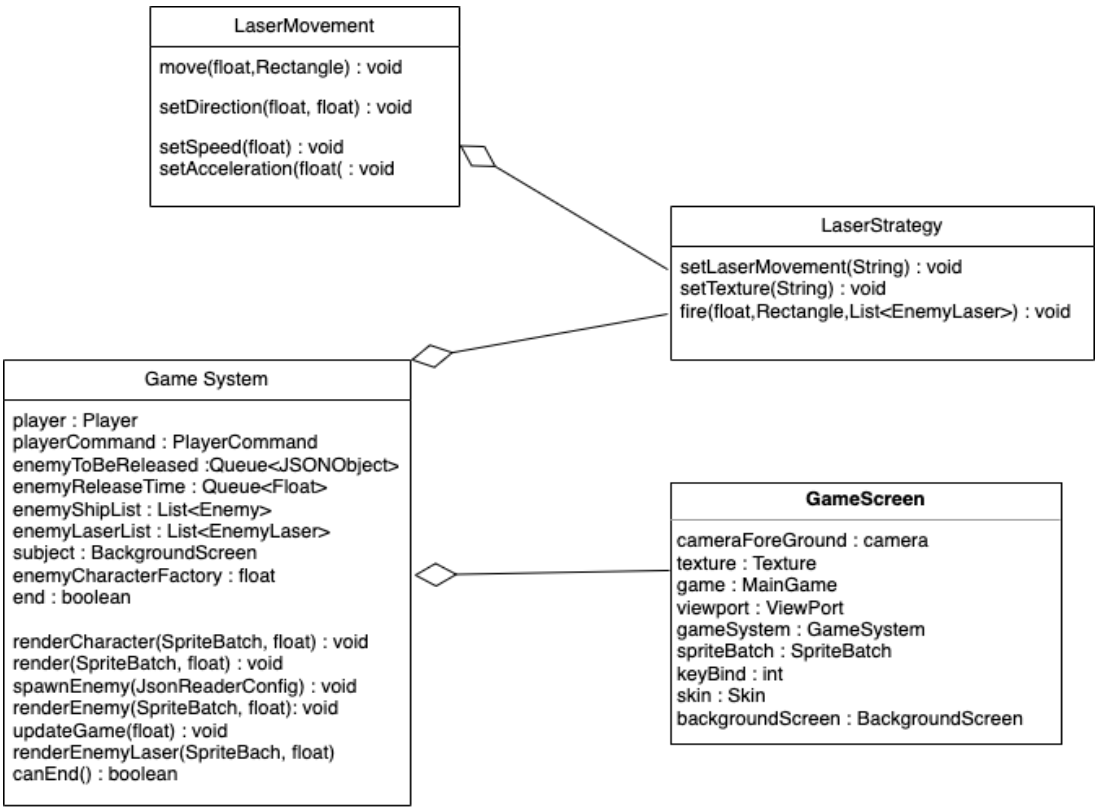

Figure 2-4. Game Controller Subsystem

2.2.4 Design Patterns

Below mentioned are the design patterns, I have used to build the application so far.

2.2.4.1 Abstract Factory Pattern for Game Objects

In the implementation, we have GameControllable class which is implemented by Player class. Then we have EnemyShipCharacter, which implements Enemy interface. Also , we have Laser strategy which is implemented by NormalLaserStrategy and then we have LaserMovement which is implemented by LShapedLaserMovement and StraightLineShapedLaserMovement. Since, these classes implement another class, they constitute Abstract Factory Pattern.

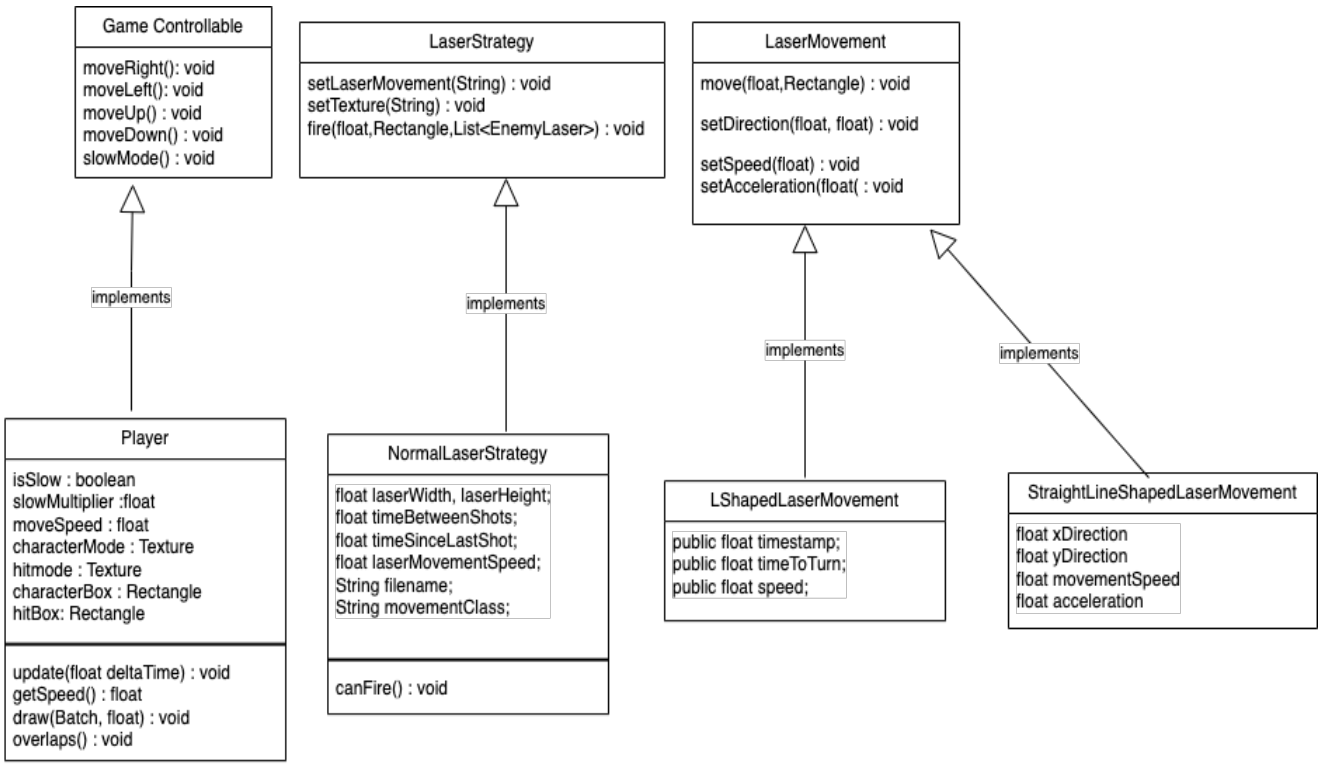

Figure 2-5. Abstract Factory Design Pattern

2.2.4.2 Factory Pattern for Game Objects

In the implementation, we have Enemy class which is inherited by EnemyShipCharacter. This compromise of a factory pattern design. Factory design pattern deals with inheritance, in which one class extends the attribute of another class and overrides its method.

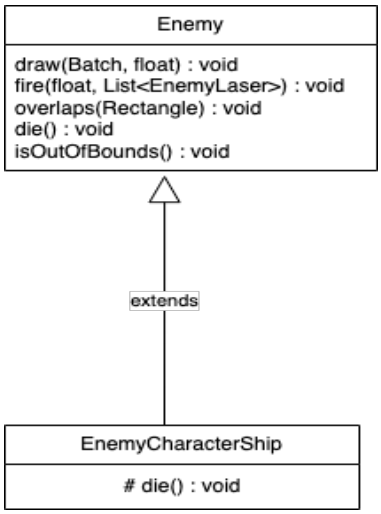

Figure 2-6. Factory Design Pattern

3. Subsystem Services

Game object subsystem holds all the data needed by Game Screen and Game controller. It basically oversees organizing and storing all the data needed by the application for modelling purpose. In Game Object subsystem, data includes all Game Objects needed for modelling the game including Enemy, The Player, Laser movement and enemy factory and Game Controllable. In addition to the mentioned classes, it includes their creational and structural design pattern. This module exposes operation which corresponds to fetch and query data parallelly abstracting operations as much as possible. One of querying data includes an operation which involves in determining whether two game objects collided with each other. In addition to that this module also contains data-related logic that the user works with and business logic data which the model uses. Game Object subsystem exists independently and has no knowledge of other two subsystems and works on operations asked by those two subsystems.

Game Screen subsystem serves as a user interface and whatever is displayed to the user is via this module. It retrieves data from Game Data subsystem processing Game Object subsystem model and displays the content to the user and user is also able to modify the data. This part corresponds to the view module of MVC framework.

Game controller subsystem handles every user request. The controller module binds Game screen subsystem and Game Object subsystem. This works on the game data to modify and update as per the user request and then to reflect the changes through Game Screen subsystem. So, in general it serves as an interface to process logic and user requests and interact with Game Screen to render the display/output.