Software Design Document

for

Mask Up

Team: Team Madison

Project: Mask Up

Team Members:
Phong Bach
Nain Galvan
Russell Heppell
Daniel Nuon
Madison Velasquez

Table of Contents

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
1	3/17/21	Began Document	
2	3/27/21	Added Multi-layer component diagram	Layout of the architecture of our project.
3	3/28/21	Added diagrams for Section 2.2.1	Screen interface class diagram
4	3/28/21	Added diagrams for Section 2.2.3	Object layer class diagrams

List of Figures

1. Introduction

Mask Up is a bullet hell desktop application game. The game's theme is inspired by the detrimental year of 2020, which felt like a bullet hell itself. The main boss is Covid, the mid boss is Karen, and the grunts(normal enemies) are Murder Hornets and Bats. There are 4 stages to the game, where the first and third stage consist of grunts. The second stage is the mid boss Karen, and stage four is the final boss Covid.

1.1 Architectural Design Goals

<u>Design Goals for Mask Up:</u>

Two quality attributes (QA) important to our system are **performance** and **testability**. These QA's are important to our game because we want the game to be reactive and ready to carry out tasks when we need to be (spawn enemies, shoot bullets, detect collision). We also want the game to be testable so bugs can be easily found and new/improved features can be quickly implemented.

Design choices we made to achieve these QA’s:

Testability QA

1. *Applied getters and setters to each class for valuable data:*
   In our abstract class Entity is where the most important data was stored, in order to protect the data and make this data available for testing, we implemented getters and setters for variables such as image, health, x-position, y-position, name, and bullet.
2. *Implemented a multi layered architecture that separate layers so lower levels can be tested first before higher levels:*
   This can be seen in figure 2.1.1. We can first test the lower-level Objects, then moving on to the Game Logic, and finish testing with the Interface layer after we make sure that the lower levels are correctly implemented.
3. *Applied high cohesion with low coupling and separation by focusing on the sole responsibility design principle:*
   This can be seen in the class controllers such as EnemySpawningController (only handles enemy spawning), EnemyMovementController (only handles enemy movement), BulletMovementController (only handles bullet movement), BulletSpawningController (only handles bullet spawning), and TimeController (only keeps track of time). We limit the complexity of the architecture elements by giving each class a specific task with limited interaction with other classes. It helps achieve controllability and observability in our code and help make the testing tractable.
4. *Limit structure complexity*

We avoid cyclic dependencies between classes and reduce dependencies between components in general. We also simplified our inheritance hierarchy. We limit the depth of the inheritance levels to the max of four levels. This max can be seen in the generalization between the classes Entity-> GameObject-> Enemy-> Concrete enemy.

5. *Abstract Data source:*
   Abstracting classes such as Enemy, GameObject, Entity, Ammo, and command allows us to substitute data easily and control input, therefore, making the game easier to test.

Performance QA

1. *Implemented singleton pattern:*
   We implement a singleton pattern in our controller classes. It increases the performance of the game because only one instance of commonly used objects are created. This can be seen in the classes Player, GameController, GameResources, and StageController.
2. *Limit event response*
   We implemented a linked list for the enemy, enemy bullets, and player bullets so we do not try to update the bullets all at the same time. We traverse and loop through the list to update each of the elements one at a time.
3. *Reduce overhead*
   In the BulletSpawningController, we continuously remove the bullets that are out of bounds of the screen. The game does not have to continue to keep track of those after the bullets have left the screen. We also implement the same thing and reduce overhead in the EnemySpawningController by removing an enemy if they reach a dead state or when they time out and leave the screen.
4. *Increase resource efficiency*
   We implemented a command pattern for enemy and player collision detection. We only check for collision if the object is close by, which reduces unneeded computation. Every time collision detection is called, new rectangular objects have to be created for the current object and other objects. It prevents the need to make new rectangular bounding boxes for other objects on the screen that are too far from the current object that is checking for collision.
5. *Reduce computational overhead*
   We implemented a GameResources object for the application that keeps track of all the resources that are used by the game objects, game logic, and the interface. A GameResources object prevented multiple computations of commonly used logic and UI values.
6. *Implemented an AssetsManager*
   Every single one of the assets is loaded in memory once and we can use them as many times as we want. This also means that if we load an asset multiple times, it will actually be shared and only take up memory once. This aided performance because it prevented multiple PNG loading of the same image.

2. Software Architecture

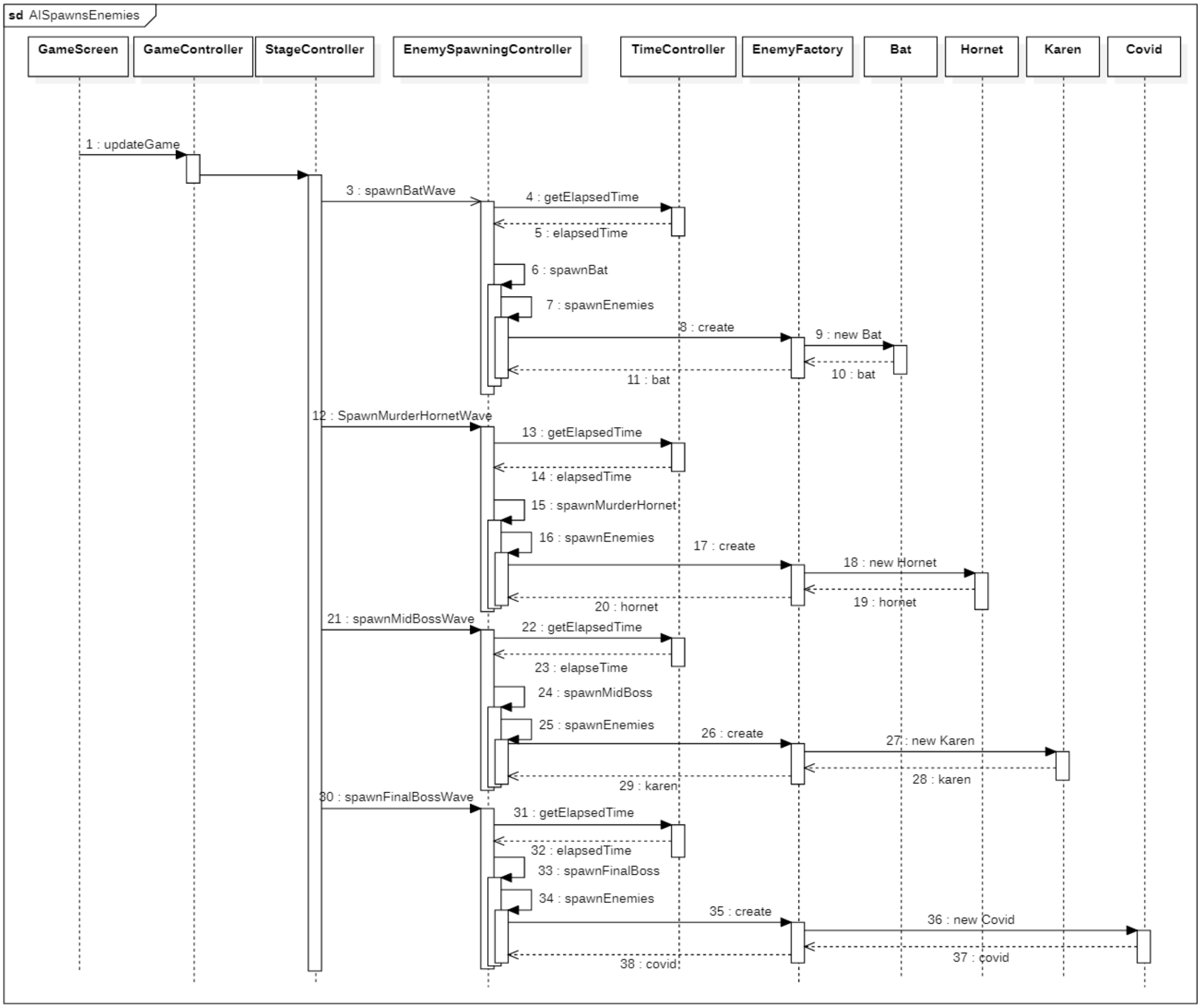

Figure 2.1: AI spawns enemies sequence diagram

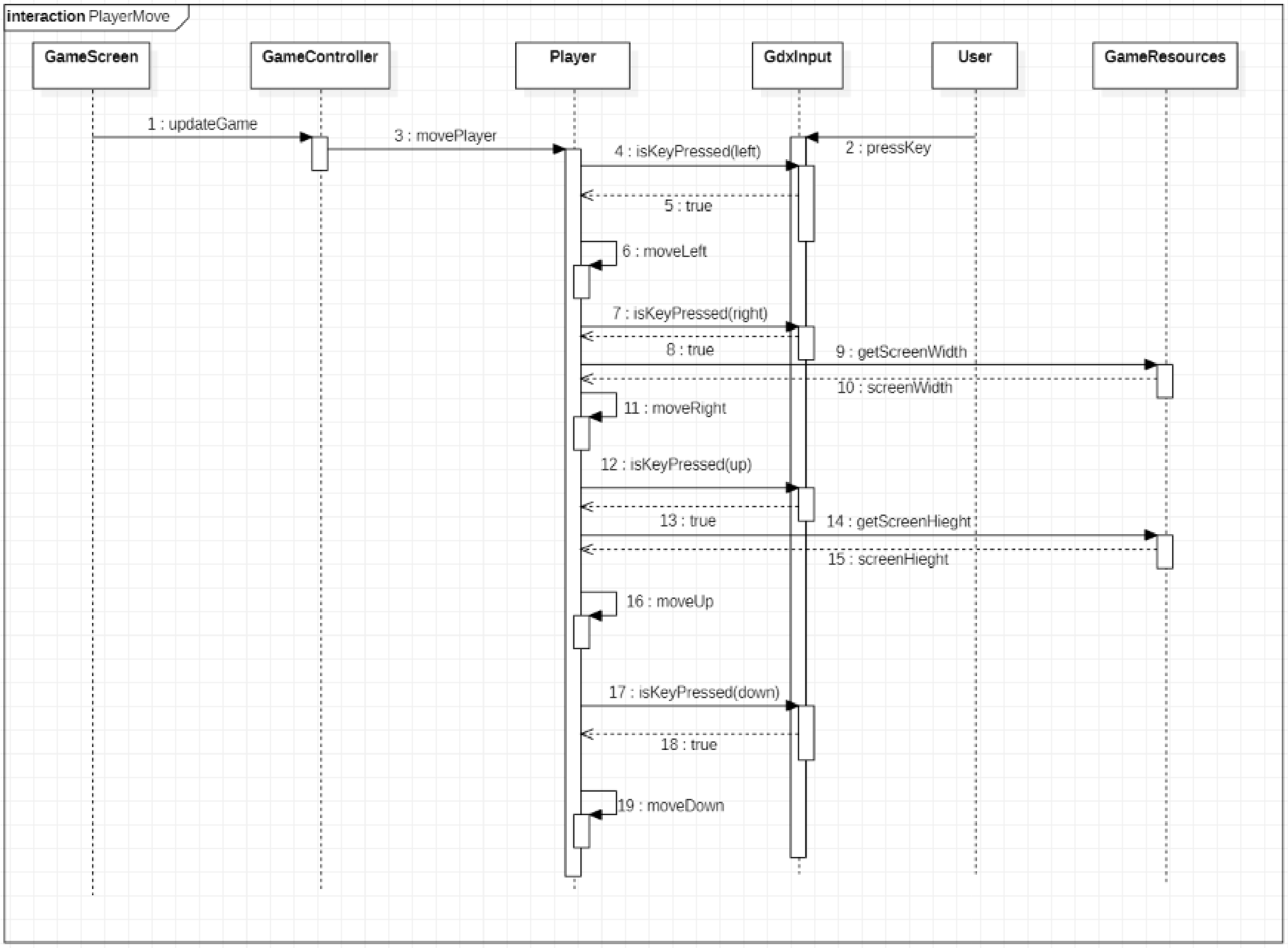

Figure 2.2: User controls player to move

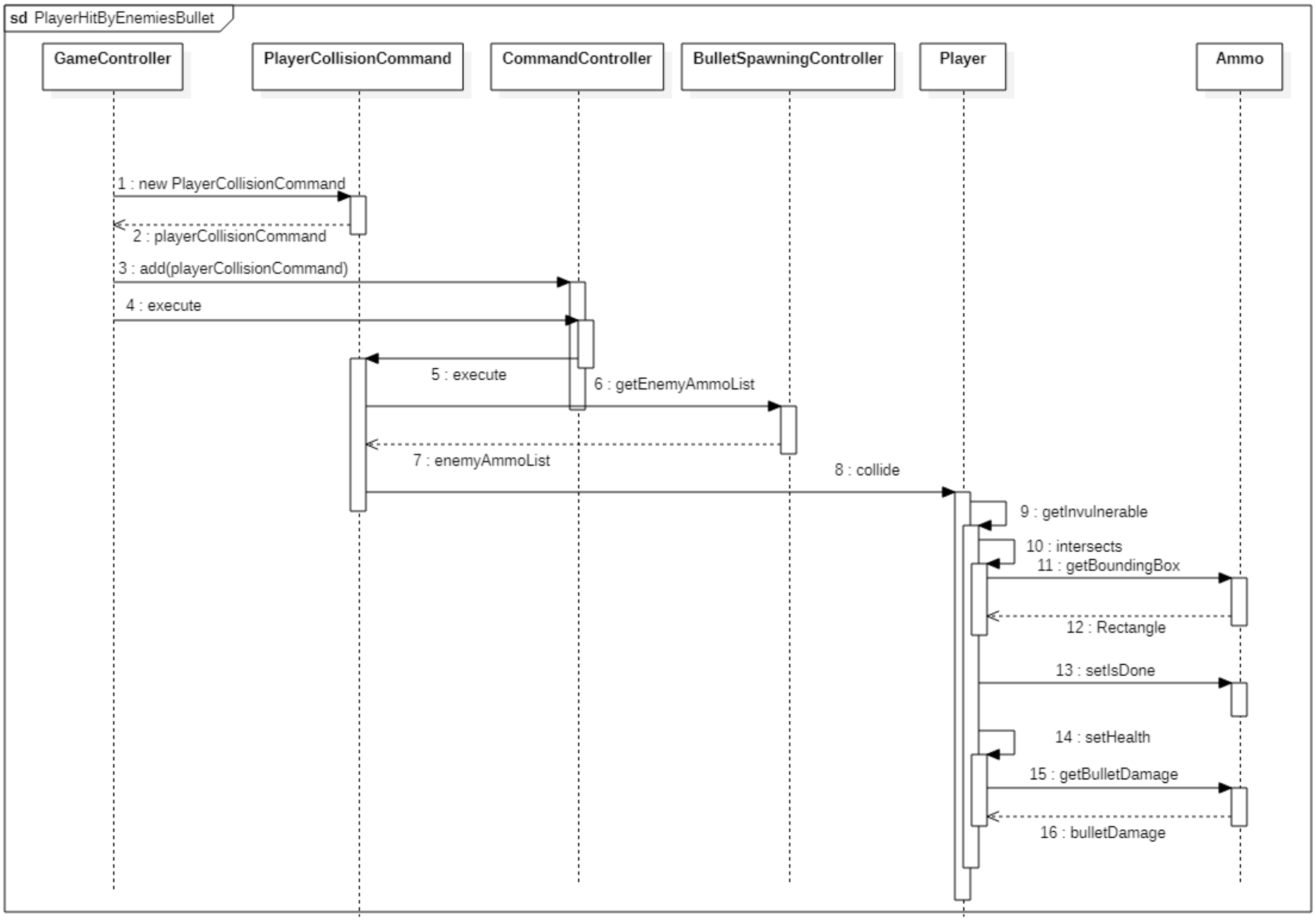

Figure 2.3: Player hit by enemy bullets

2.1 Overview

The architectural pattern chosen for the design model is a Multi-Layered architecture, consisting of three layers: Interface, Game Logic, and Objects. The Interface layer consists of the boundary objects, which interact directly with the user. The Game Logic layer deals with the control of objects, game processing, and overall logic. The Objects layer consists of the storage of objects, representing different entities in the project. The rationale behind choosing this pattern is that it decouples data access, data storage, and data presentation. This allows for extension and addition of new components to the subsystems, and lower layers do not need to be modified to compensate for changes.

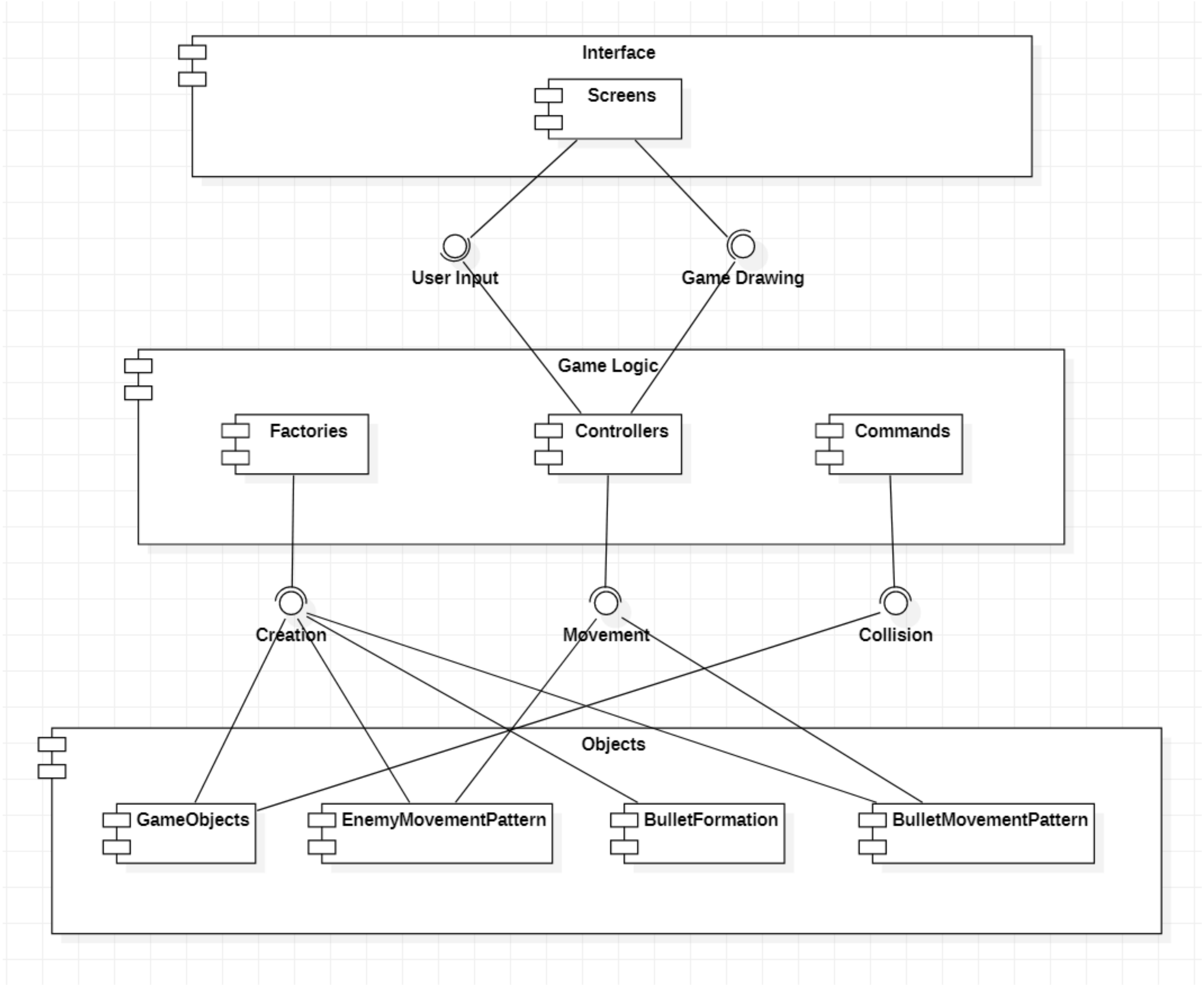

Figure 2.1.1: Multilayered architecture of the project

2.2 Subsystem Decomposition

For this project, the subsystems include Interface, Game Logic, and Objects. This subsystem decomposition maps directly to the layered architecture. Our Interface layer consists of our screen classes, which the user directly interacts with. The Game Logic layer contains our various factories for objects, controllers for various tasks (spawning, enemy movement, time, etc.), and our command for collision detection. The Object layer contains all of our game objects(enemies, bullets, player, etc.), enemy and bullet movement patterns, and the bullet formation classes.

2.2.1 Interface

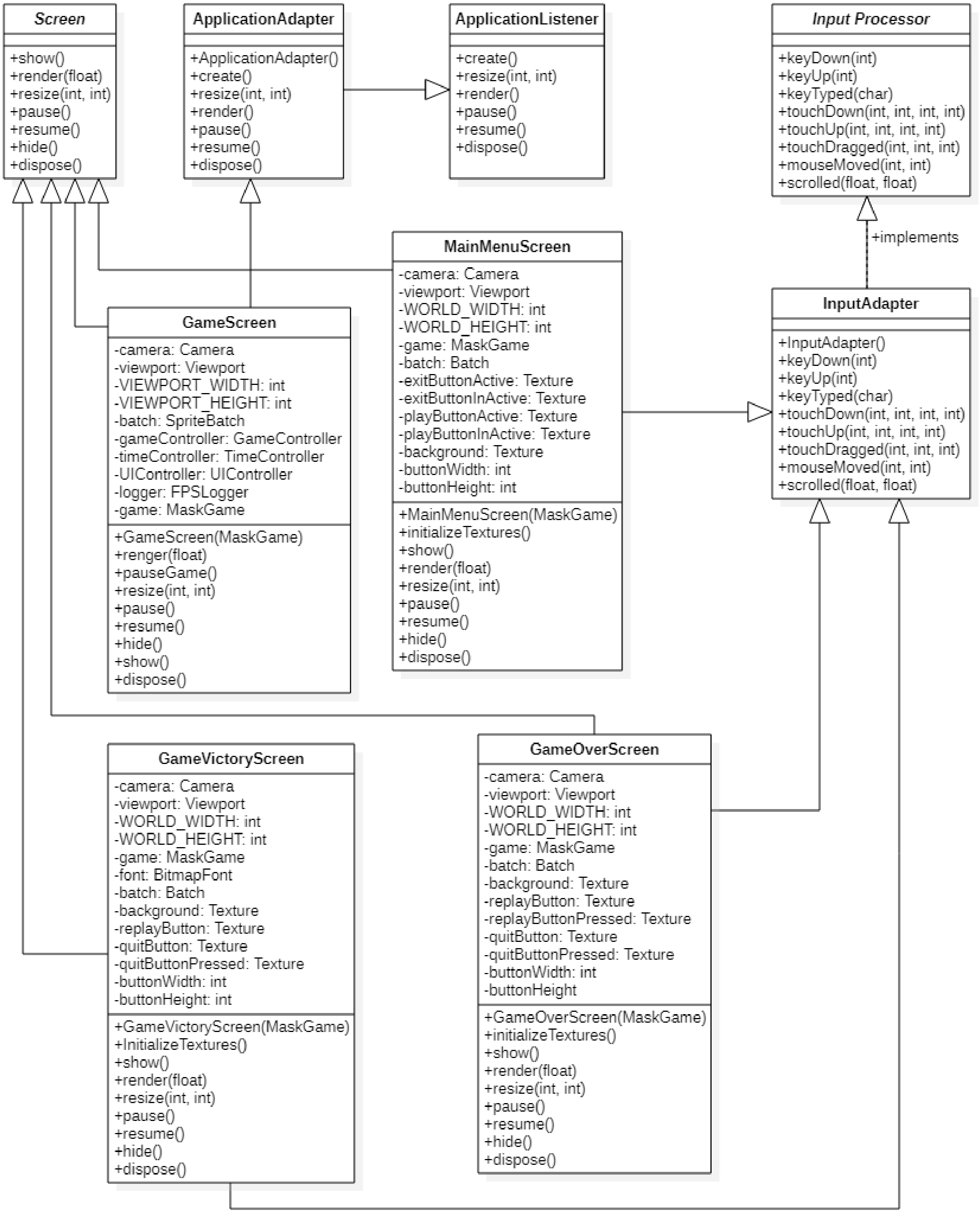

Figure 2.2.1.1 Screen Class Diagram

Our current Screen classes include GameScreen, GameOverScreen, GameVictoryScreen, and MainMenuScreen. As the game goes through stages, different screens will be set from the GameController class based on input from the user, or some sort of event from the game. These screen classes are what interact directly with the user, and they do not handle any game logic.

2.2.2 Game Logic

The Game Logic subsystem contains the many factories we have for creating objects, following the Factory Design Pattern. This layer also has our various controllers, including TimeController, StageController, EnemySpawningController, BulletSpawningController, Movement Controllers and ScoreController. Collision detection follows a Command Pattern, for the player colliding with enemy bullets and enemies colliding with player bullets. These are the classes doing all the internal work for the processing of the game. All logic is contained in this layer, interacting with the layers below and above it.

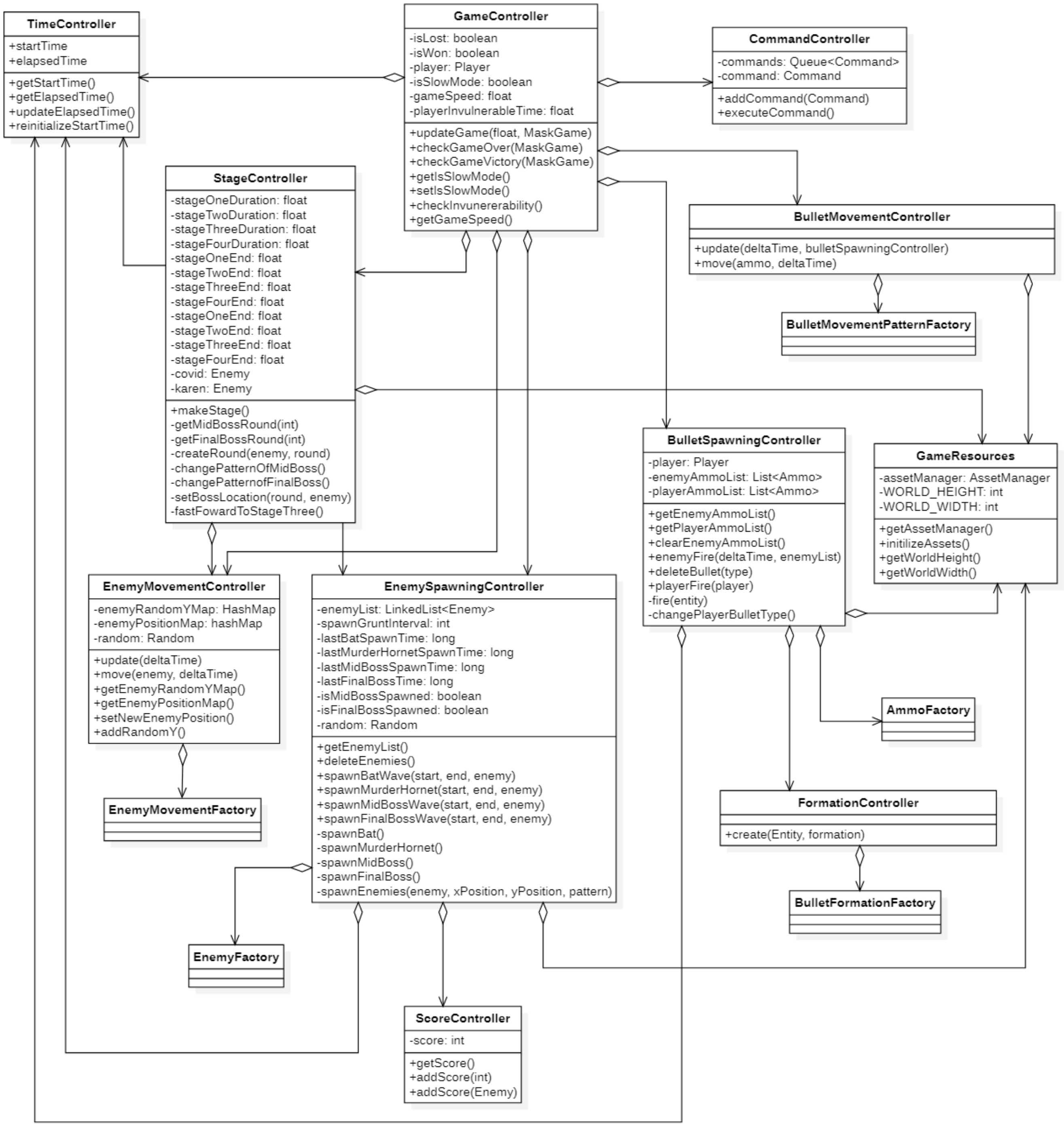

Figure 2.2.2.1 Game Logic Class Diagram

2.2.3 Objects

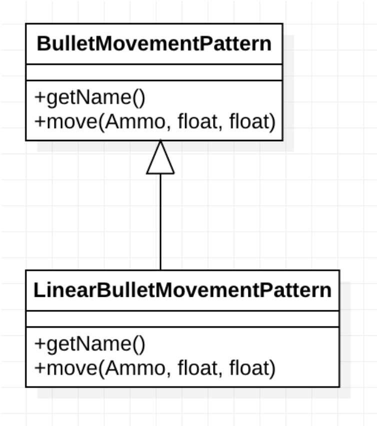
BulletMovementPattern
+getName()
+move(Ammo, float, float)
LinearBulletMovementPattern
+getName()
+move(Ammo, float, float)

Figure 2.2.3.1 Bullet Movement Pattern Class Diagram

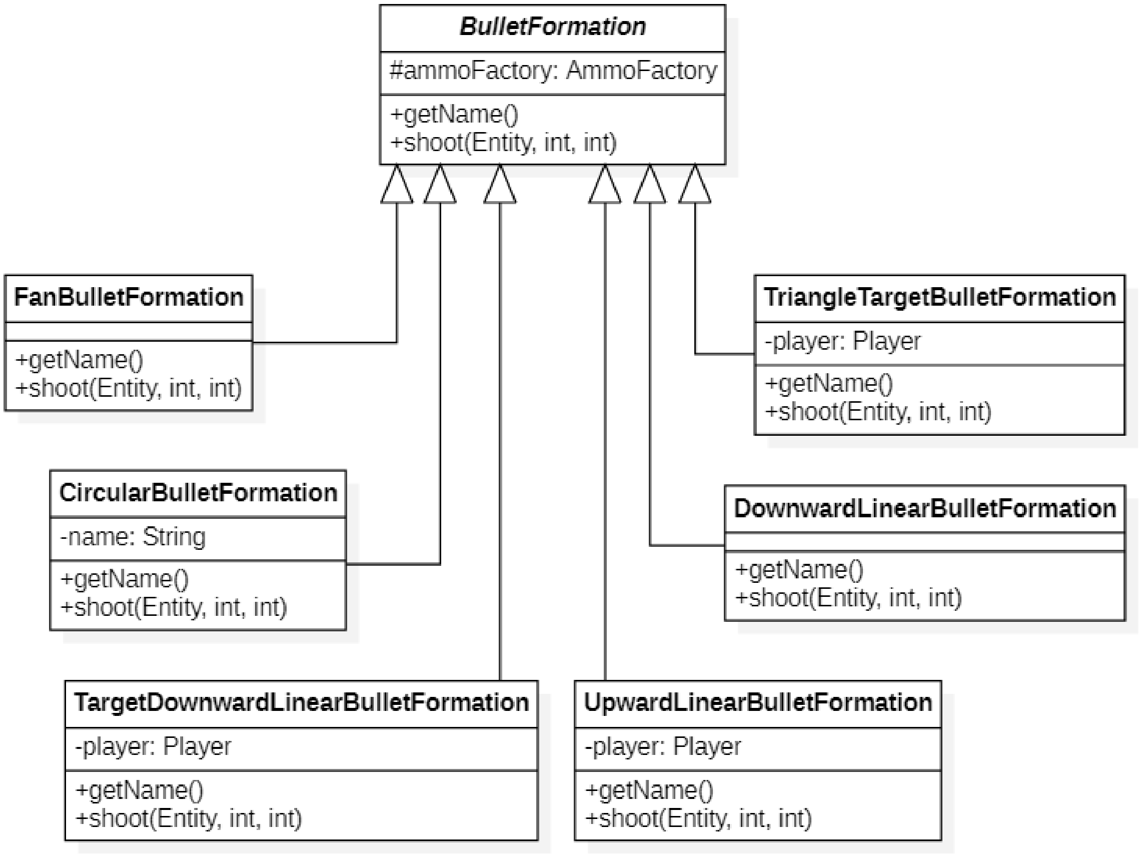

Figure 2.2.3.2 Bullet Formation Class Diagram

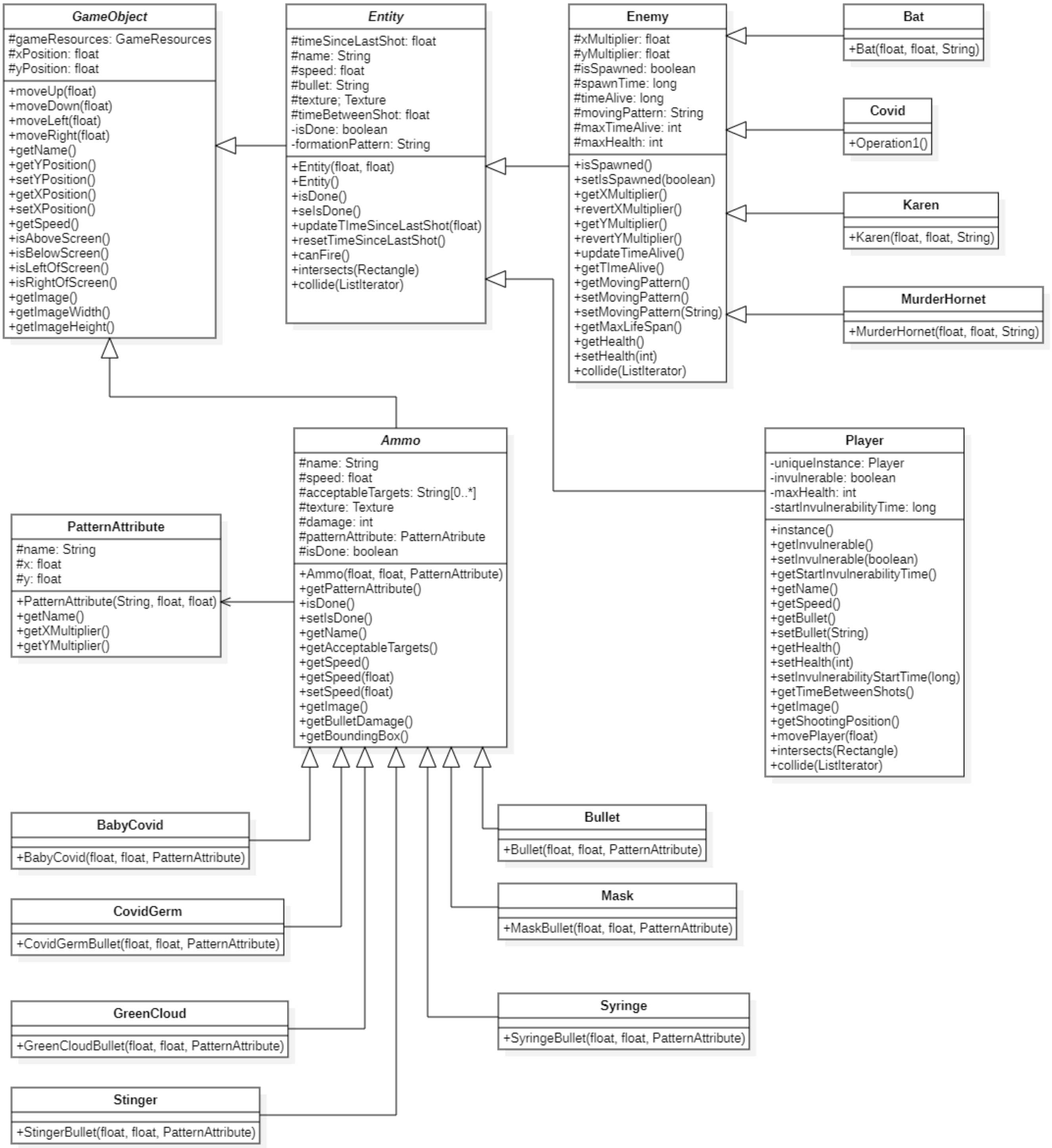

Figure 2.2.3.3 Game Object Class Diagram

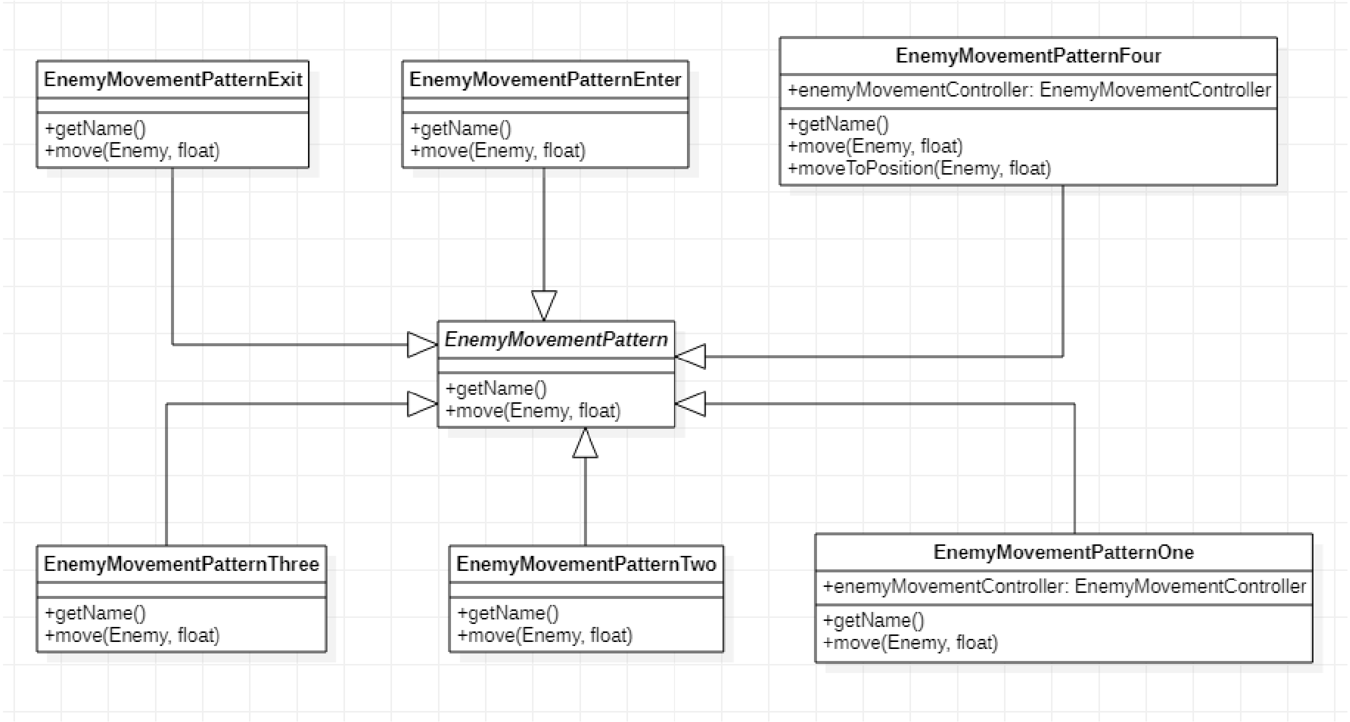

Figure 2.2.3.4 Enemy Movement Class Diagram

The Objects subsystem includes our definitions for game objects, enemy and bullet movement patterns, and bullet formations. These classes store the information pertaining to the objects, providing retrieval, and querying of object attributes so the layer above can perform logic and control. They also provide the interface for collision detection, so knowing when proper bullets and entities collide can be performed in the command controller.

2.2.4 Design Patterns

Design Pattern One: Factory Pattern

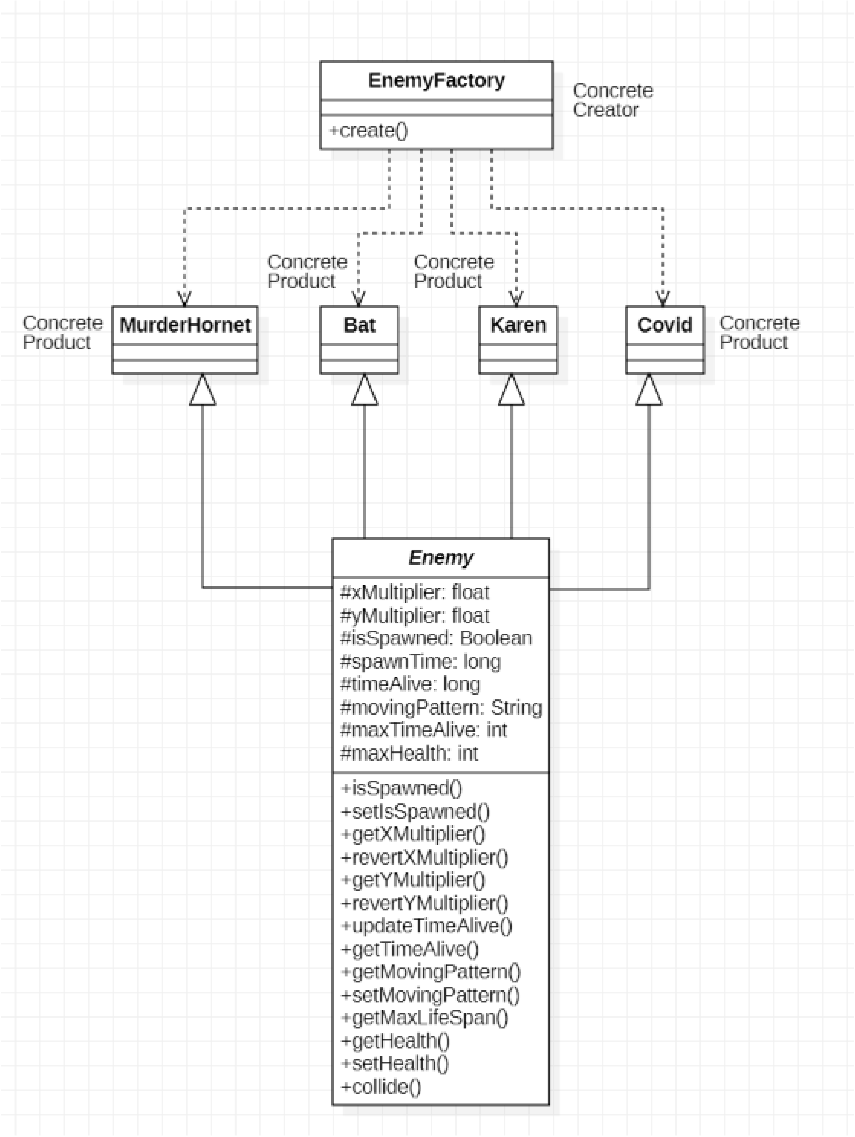

Figure 2.2.4.1 Factory Pattern Class Diagram

Design Pattern Two: Singleton Pattern

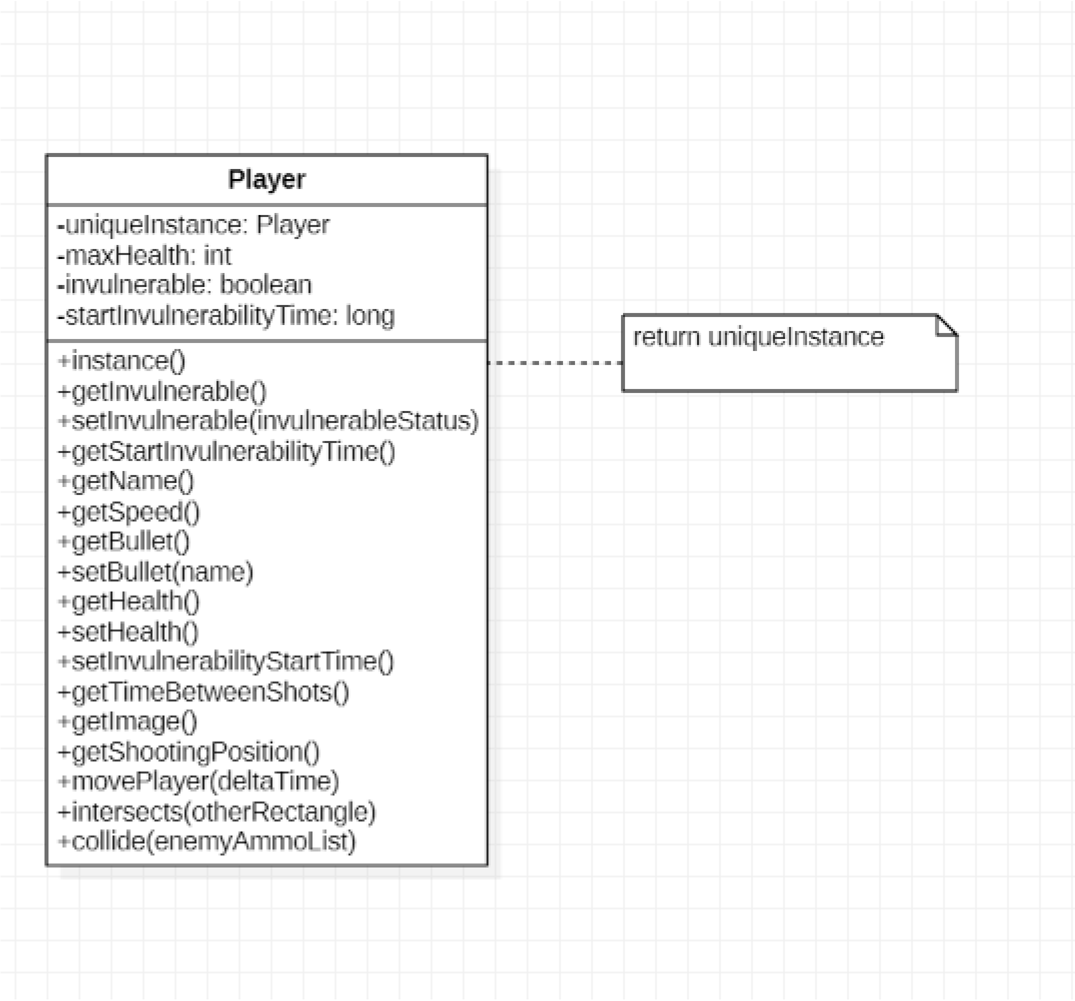

Figure 2.2.4.2 Singleton Pattern Class Diagram

Design Pattern Three: Command Pattern

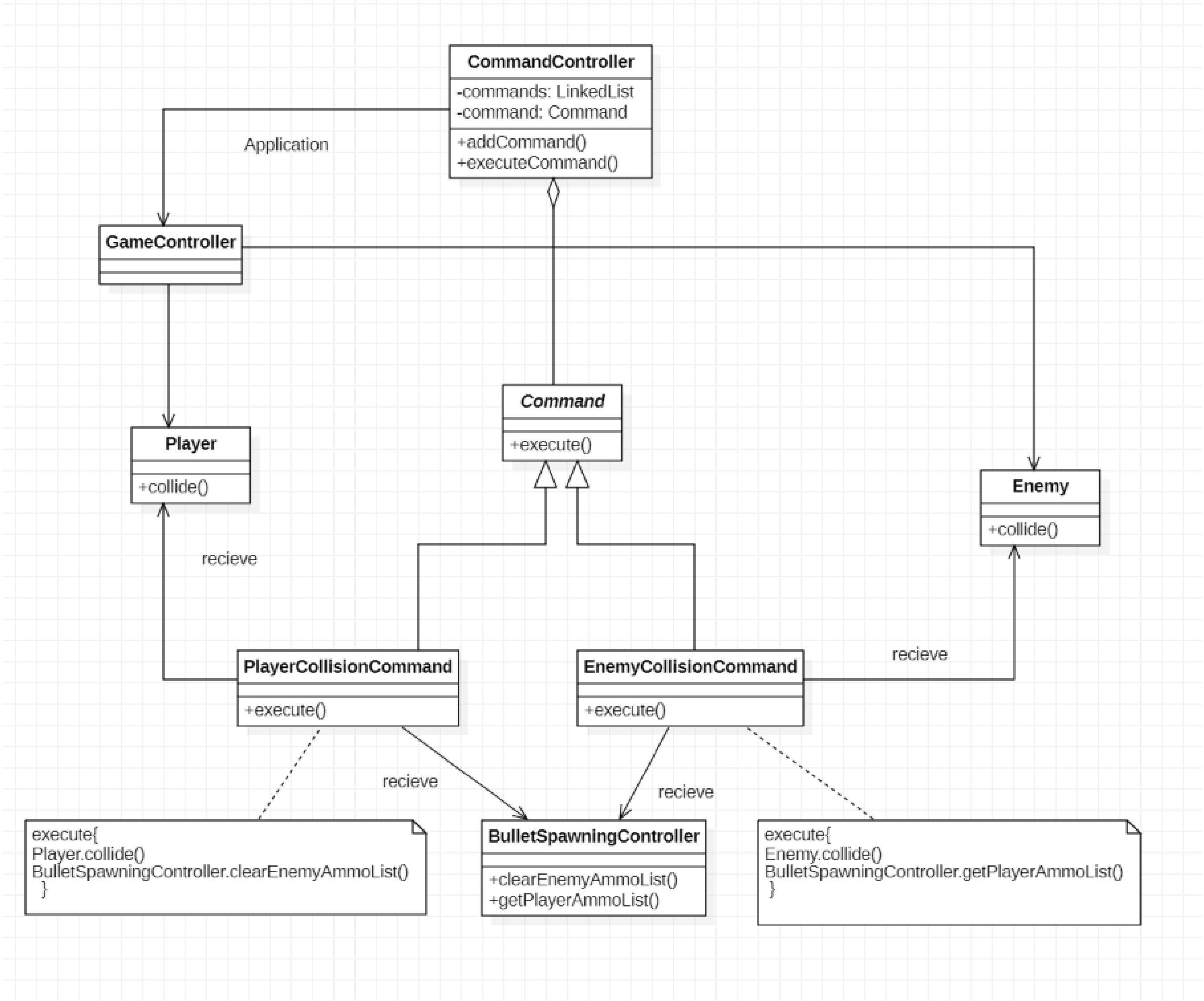

Figure 2.2.4.3 Command Pattern Class Diagram

3. Subsystem Services

Dependencies between Interface and Game Logic

1. The Interface must provide to the Game Logic any user input. This is necessary so when the user gives input, the Logic layer does the correct calculations/changes to accurately reflect what the user should see.
2. The Game Logic must provide game rendering to the Interface. Inside of each screen class, the render() function acts as main, and all of the proper logic controllers are called to perform game processing. One of the controllers, the UIController, is in charge of drawing all objects onto the screen. This way, the screens do not deal with the drawing of objects, and the logic layer can provide drawing capabilities.

Dependencies between Game Logic and Objects

1. Objects need to provide implementation capabilities so factories in the Logic layer can create them. Following the Factory Design Pattern, object creation is abstracted out so creation is hidden, and the overall system is independent of instantiation. It is crucial that objects allow for external creation.
2. Objects need to provide movement capabilities so controllers can manipulate and control objects. This includes providing ways to access and set coordinates of objects, move functions, and ways to check for boundary conditions (above screen, below screen, etc.).
3. Objects need to provide collision detection so the command can control collision checking.

User Input - Provides the user input for the Logic layer.

Game Drawing - Provides drawing capabilities for the screen to show the game.

Creation - Provides creational capabilities so the Logic Layer can control creation of various objects and entities.

Movement - Provides movement capabilities and necessary attribute retrievals so Logic Layer can manipulate and control objects.

Collision - Provides collision capabilities so the Logic Layer can control the detection of collisions between Player and enemy bullets, and between enemies and player bullets.

See figure 2.1.1 for component diagram.