Software Design Document

for

Bullet Hell

Team: Team Blaster

Project: Bullet Hell Shooting Game (BH-STG)

Team Members:

Matthew Bauer

Charlie Wong

Trevor Naze

William Hiat

Table of Contents

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
01	03/24	Initialized the document and created the framework of sections.	Initial version
02	03/25	Created section 1	Added section
03	03/26	Created section 2	Added section
04	03/26	Added diagrams and subsystem descriptions	New UML diagrams

List of Figures

List of Tables

1. Introduction

1.2 Architecture Design Goals

From a game programming and technical planning perspective, it is important to identify and discuss system/design qualities, objectives, and non-functional requirements to ultimately enhance the game's fun factor. Quality attributes such as availability, and testability play a significant role in software systems as a whole by ensuring product ease-of-use, and user satisfaction.

When considering the *availability* of a system, we can consider the following tactics:

- Detect faults
- Recover from faults
- Prevent faults

The “recover from faults” tactics are refined into “preparation-and-repair” tactics and “reintroduction” tactics. In “preparation-and-repair tactics”, adhering to *software upgrades* allows the delivery of new features and capabilities to the program, as well as class patches to deliver bug fixes. Adopting this tactic is relevant to game design/development i.e., changing game rules or algorithms, deploying patches to fix compatibility issues.

In addition, when considering the *testability* of a system, we can consider the following tactics:

- Control and Observe System-state
- Limit complexity

In the “control and observe system-state” tactics, utilizing a *sandbox* provides unconstrained experimentation/testing on a particular instance of a system without permanent consequences. With an isolated testing environment, adopting this tactic enables testing of new programming code without affecting the application, system or platform on which it runs; this is a widespread quality in video game design/development.

2. Software Architecture

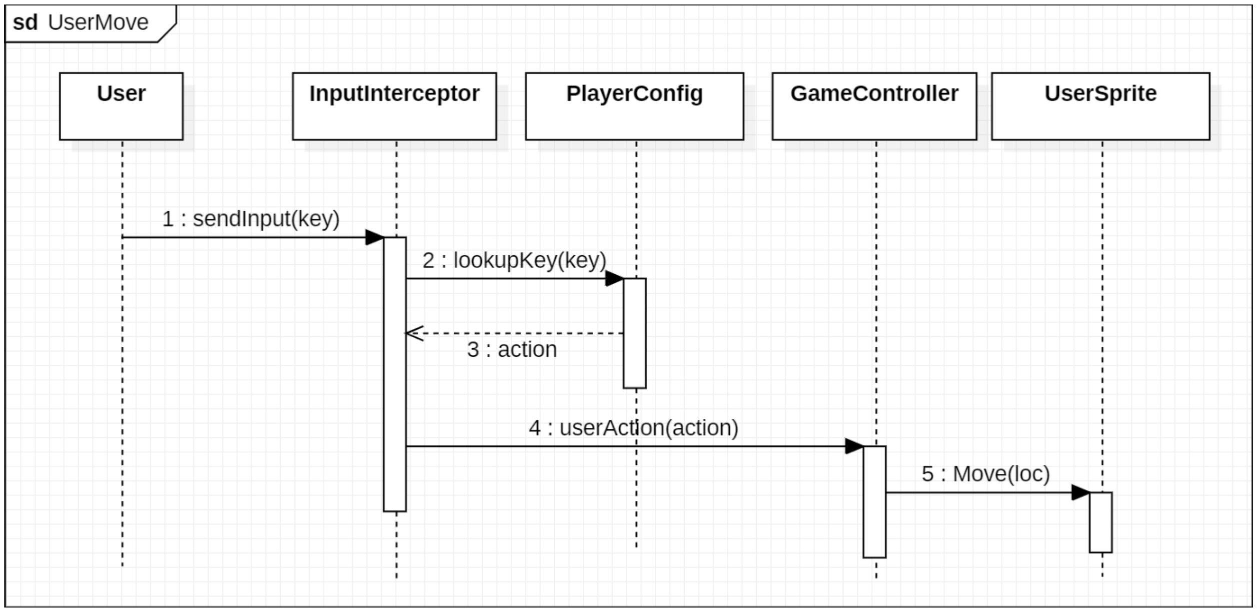

Fig 2.0.1. Sequence Diagram for User Movement

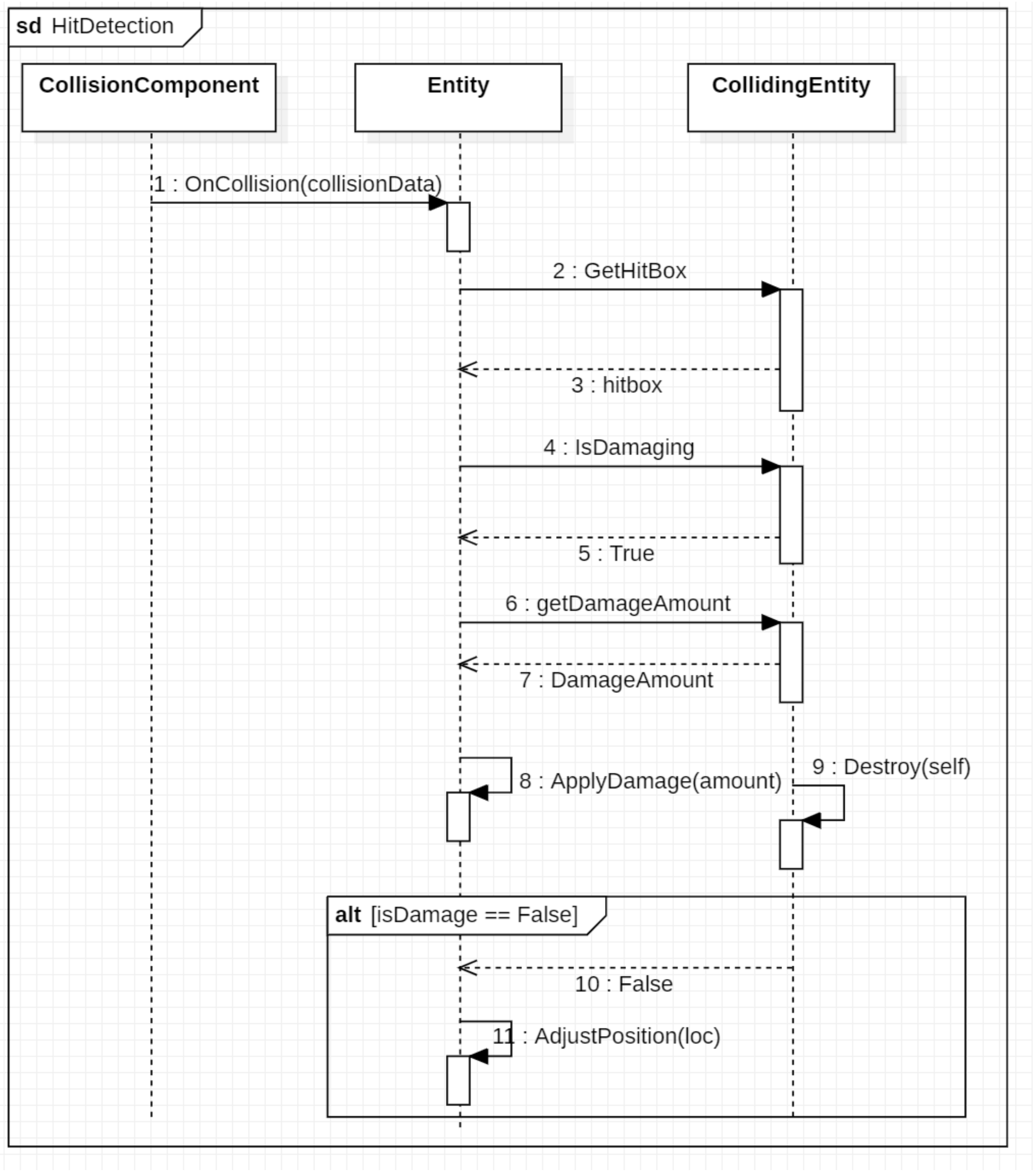

Fig 2.0.2. Sequence diagram for entity hit detection used by both players and enemies

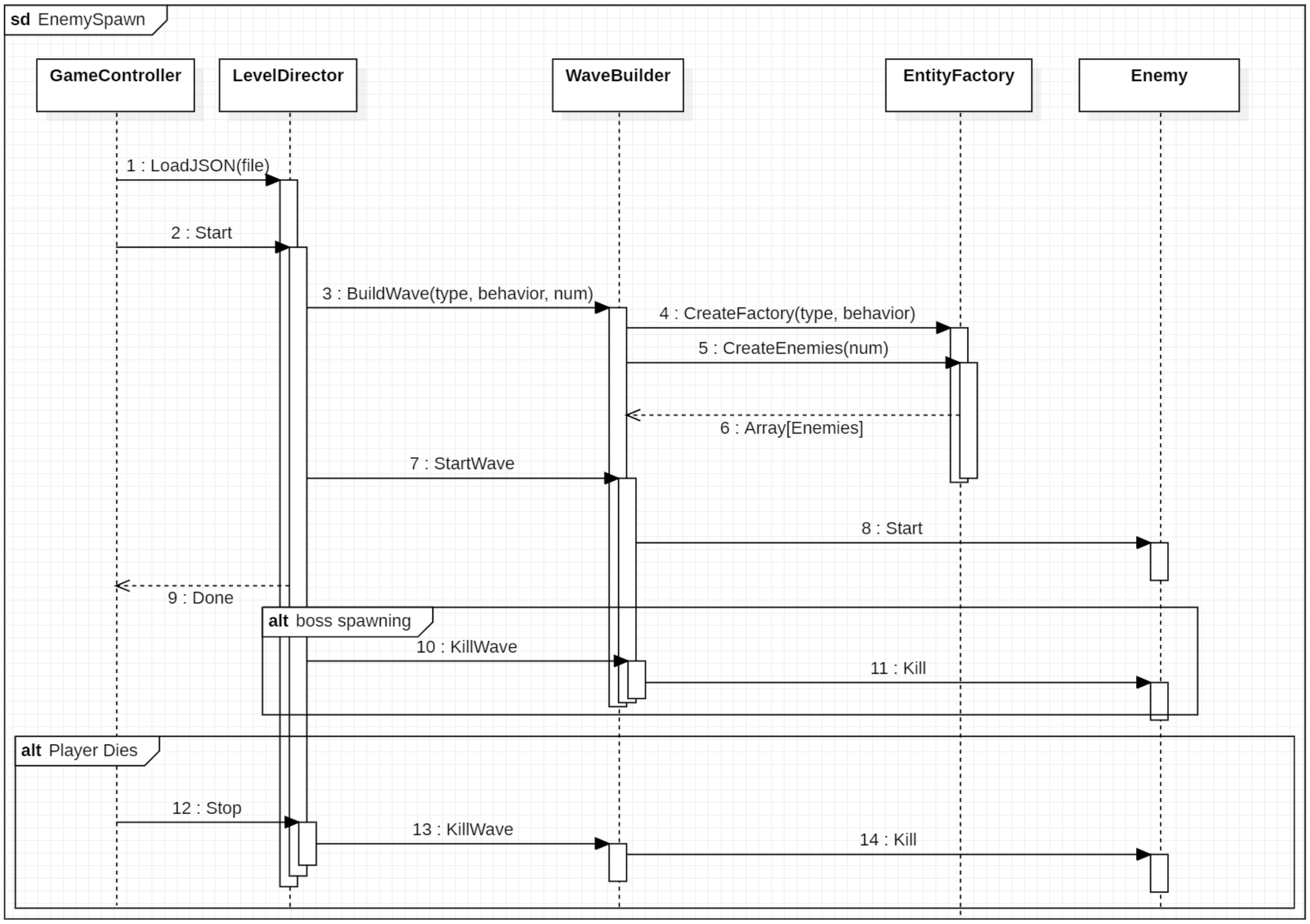

Fig 2.0.3. Sequence diagram for enemy wave generation

2.1 Overview

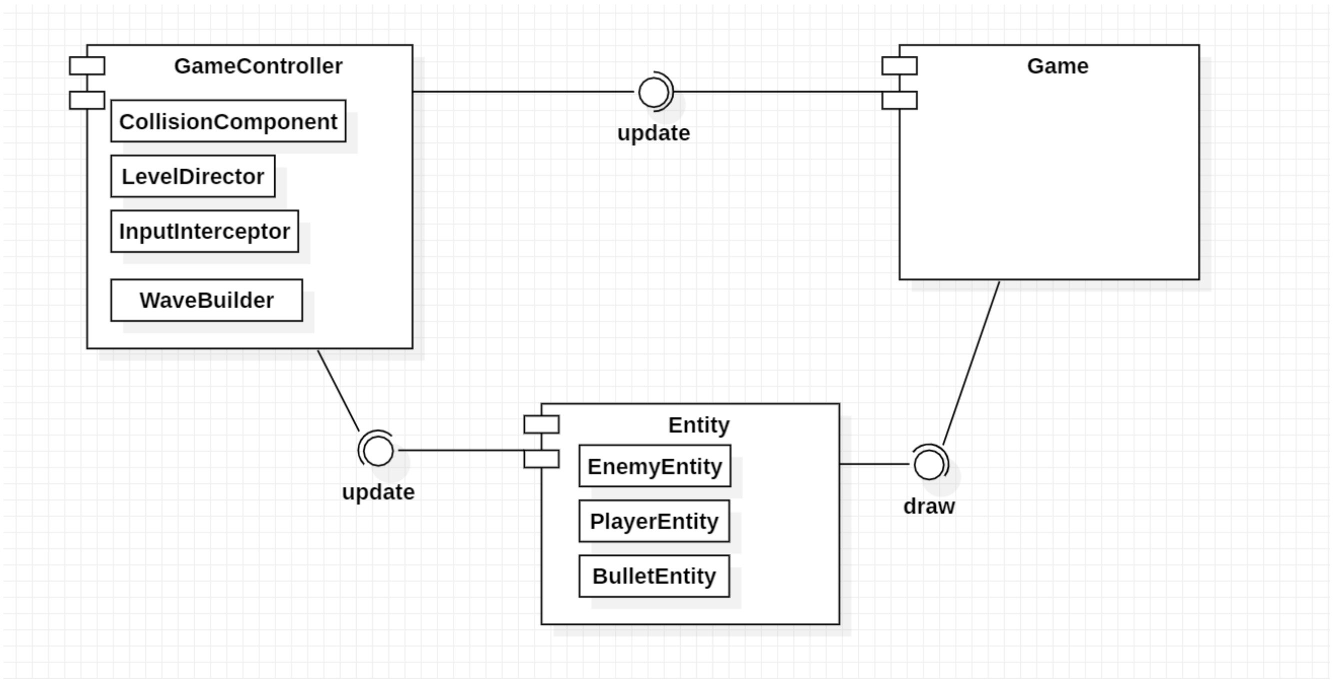

Figure 2.1.1 Software Architecture for Bullet Hell

Bullet Hell utilizes a Model-View-Controller (MVC) pattern for its architecture design. This design pattern was chosen due to the design of the underlying library used for the application. Monogame's primary interface is the Game class, which provides independent functions for updating game logic and drawing the screen, which act as similar interfaces to the behaviors that a view controller would have within the MVC pattern. Within the software architecture, the Game component is provided by Monogame and acts as the view subsystem. GameController represents the Controller element of the MVC pattern, and the individual Entities collectively represent the model component of the MVC pattern.

2.2 Subsystem Decomposition

2.2.1 Game Subsystem

The Game subsystem is responsible for all the user-experience related elements of the application, as well as being the main entry point for the rest of the application logic. The Game subsystem is predominantly built around the Monogame *Game* class, which is responsible for initializing the game, loading and unloading of all assets from the game, as well as calling the periodic logic update and screen draw functions that interface with the other two subsystems.

2.2.2 GameController Subsystem

The GameController subsystem handles all simulation logic that occurs within the game. It is composed of multiple sub components including the collisionComponent, which handles all entity collisions within the simulation in a grid-based collision calculation, the Level Director, which is responsible for the timing and construction of enemies throughout the wave, the Input interceptor, which handles all user input and translates key presses into appropriate actions, and the WaveBuilder, which are instances generated by the level director to handle the creation and behavior initialization for each wave.

2.2.3 Entity Subsystem

The entity subsystem represents the meta collection of all entities within the game, including the player, any enemies and bosses, and all bullets that any of the previously listed entities generate. Each entity stores its internal state information, as well as contains calls required for updates that can be triggered by the game controller's periodic logic updates, or draw calls requested by the Game subsystem.

2.2.4 Design Patterns

Bullet Hell utilizes many design patterns in order to improve cohesion and lower overall coupling. Patterns implemented include Factory, Builder, and Singleton patterns. Some examples are included in the following subsections.

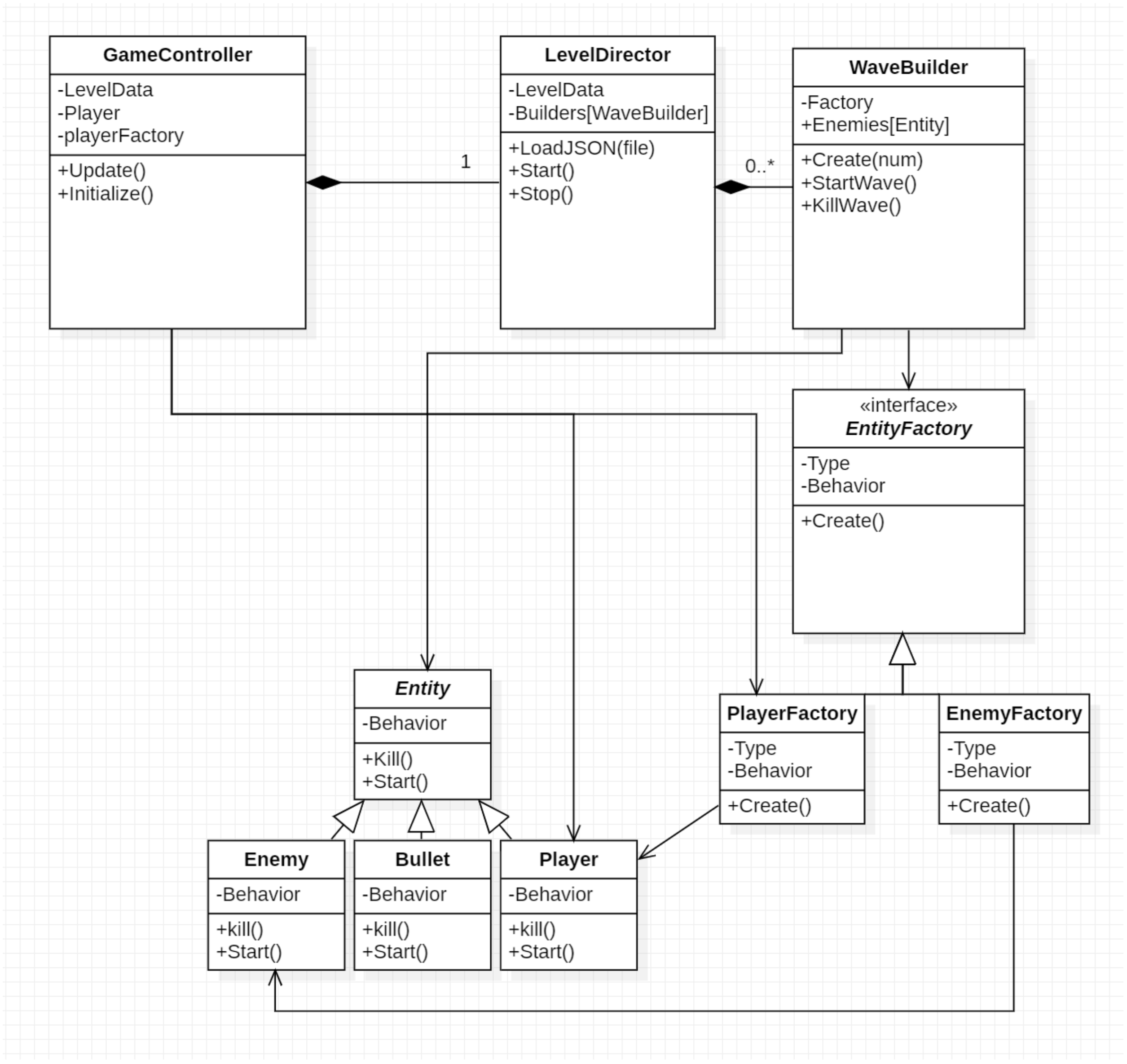

Fig 2.2.4.1. Class Diagram for Level Director logic

The Level Director subcomponent of the game controller demonstrates usage of both the builder and factory creational design patterns. Since the data for each wave defines traits such as the quantity of enemies, and their specific behavior for the wave, it is necessary to design the wave generation to allow for flexibility. This is being done by breaking the construction and management of enemies into two distinct steps. Since enemies need to be spawned sequentially in a specific order, with specific timing, the responsibility of when to deploy them closely matches justification for when a builder pattern makes sense to use. However, the creation of individual waves has no variation between the individual enemies in the wave, thus it makes sense to utilize a factory for producing many at the same time. Because of this, the implementation of enemy wave generation relies on a builder, which will instantiate factories to construct waves, and then order those factories to deploy their produced instances at specific times. This double-purpose of the Director class would normally violate the builder creational pattern, but when viewed from a behavioral perspective, deployment of instances is an additional step in the instancing process from the perspective of the application's logic.

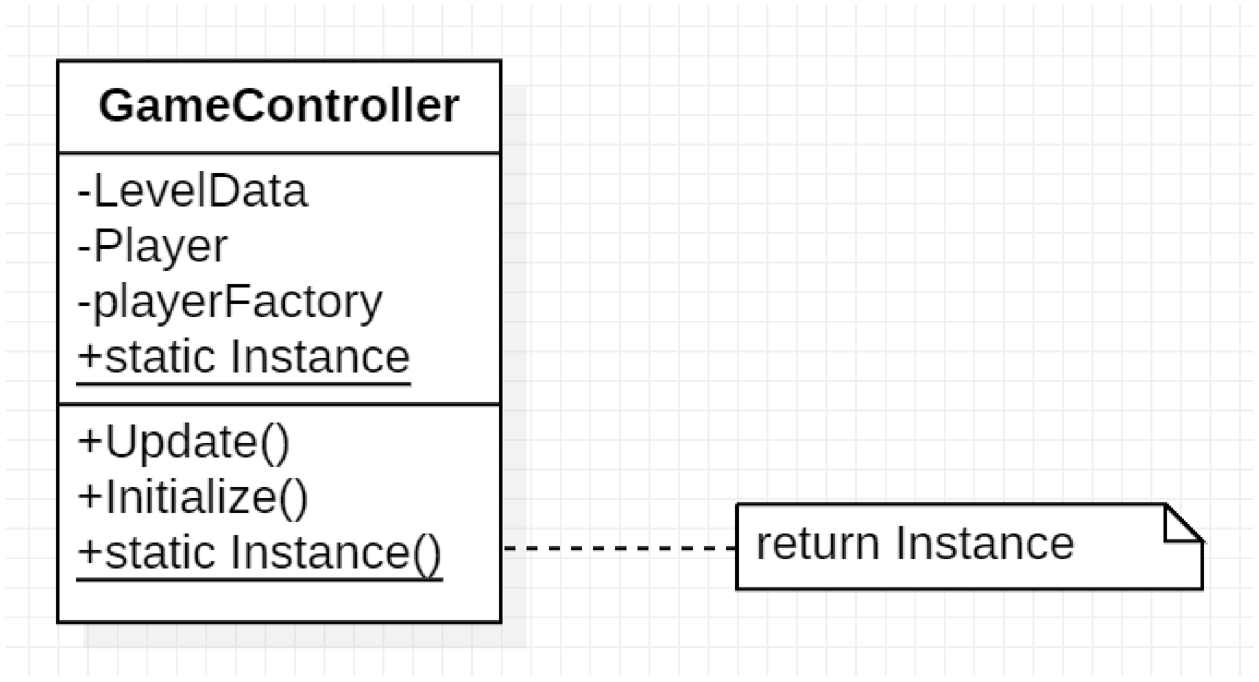

Fig 2.2.4.2. Class Diagram for GameController singleton behavior

The GameController element represents the controlling subsystem of the MVC architectural pattern, and as such, it is critically important that during all periods of operation, only one instance of the GameController class exists. Future implementation may also require direct callbacks to the game controller from elements within the entity collection (such as upon the player dying), and as such, having a single instance makes it easier to implement any dependency injection that may be required by these elements.

3. Subsystem Services

Bullet Hell incorporates various dependencies/services among subsystems (GameController, Entity, Game) in order to coordinate workflow, and manage resources.

The Entity subsystem provides (implements) dependencies for both the GameController and Game subsystems; responsibilities include behaviors and/or creation of various entities in the game. A *draw* service is provided from Entity subsystem to Game in order to display appropriate entity objects to the game screen. In addition, an *update* service is provided from Entity subsystem to GameController to update the logic of all created entities.

The GameController subsystem implements dependencies for the Game and Entity subsystems; responsibilities include handling the overall game and entity logic. An *update* service is a required interface from the GameController subsystem to Entity subsystem in order to properly build waves with the correct entities (player, enemies, bullets). The GameController provides an *update* interface to the Game subsystem in order to update the game/game logic in real time.

The Game subsystem has dependencies for the GameController and Entity subsystems; the Game exhibits the overall logic between GameController and Entity components. Opposite to the GameController, The Game subsystem requires an *update* interface in order to properly initialize/load different aspects of the game. Similarly, the Game subsystem requires an interface *draw* in order to display entity objects to the game screen. Refer to **Figure 2.1.1** for the component diagram.