Software Design Document

for

Bullet Hell: Cats vs. Dogs (CvD)

Team: Epic Gamer Game Studios (EGGS)

Project: Bullet Hell: Cats vs. Dogs (CvD)

Team Members:
Mark Shinozaki
Bryce Moser
Caitlin Cunningham
Tanner Blanken

Table of Contents

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
1.0	3/24/2024	Iteration of document for submission	

List of Figures

1. Introduction

Our project, CvD, is a bullet hell game developed using C# and the MonoGame engine. The gameplay revolves around controlling a player character around a two-dimensional space, avoiding projectiles from incoming enemy waves and shooting at these enemies to score points. CvD aims to have a modular level interpreter for users to be able to create their own levels.

1.1 Architectural Design Goals

Performance:

For our game, performance is one of the more important design qualities since we want our game to be very responsive and not lag. The last thing that you would want in a fast-paced game is any sort of slowing down of game logic since it will disrupt the player's overall game experience. To this end, we mainly want to focus on the processing time of the functions required for the game to be played, which includes the level interpreter functionality of calculating what enemies to spawn, how long the waves last, etc. To improve performance, our level interpreter consists of reading a JSON file, which then is parsed and passed into functions relating to each element of the JSON, including but not limited to enemy generation, player lives, and wave duration. To manage resources so this is done effectively, calculations will be done step-by-step with each feature of the game environment to ensure those resources are available for more high-priority tasks. For example, the theme of the level and the data about the player will be loaded first before we deal with any enemy wave processing.

Testability:

Being able to test each component within the software design of our game is imperative to a less painful development workflow as well as making sure we can find bugs quickly and then make any necessary changes to fix these bugs. There are a couple tactics relevant to the testability of a piece of software: controlling and observing the system state, and limiting the complexity of the system by reducing coupling and nondeterminism. In the case of our game, we can implement state control testing in one way by using a test JSON file to ensure that the behavior of the enemies and the level are as expected. Limiting complexity can be handled by reducing the coupling between each component in our overall software system by making sure each subsystem has as little information as it needs for its functionality, for example, the level interpreter only really needs to know that it has to spawn an enemy of a certain type, and none of the actual information about the enemy type such as its health or firing pattern etc.

2. Software Architecture

Figure 2-1: Sequence diagram for AI spawning enemies

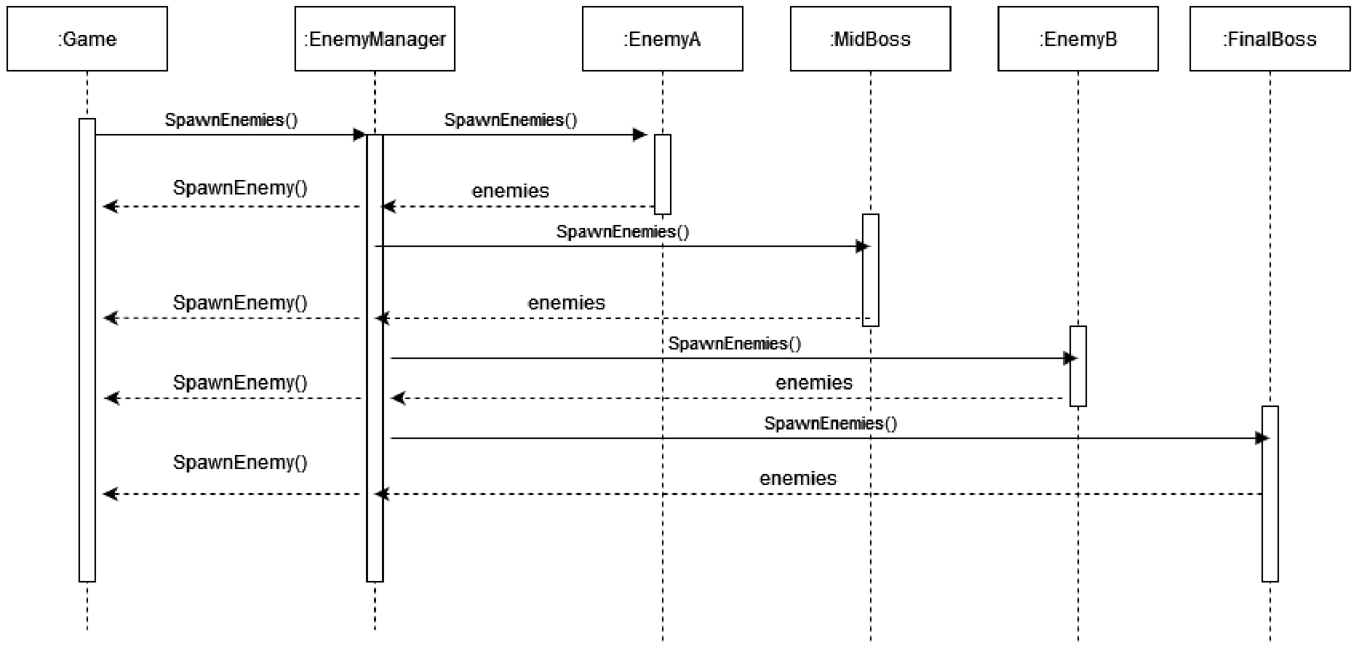


Figure 2-2: Sequence diagram for player movement control

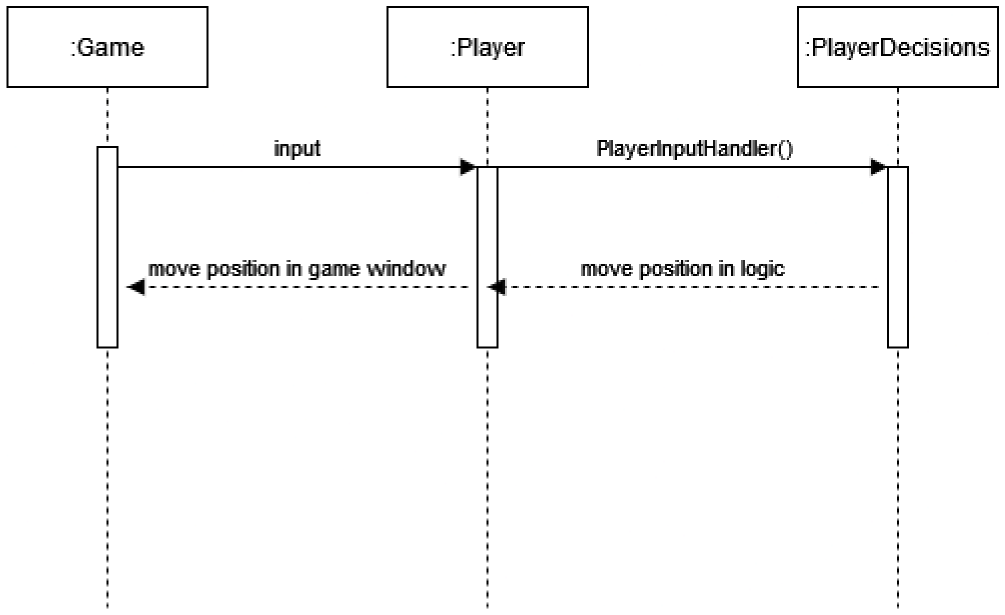
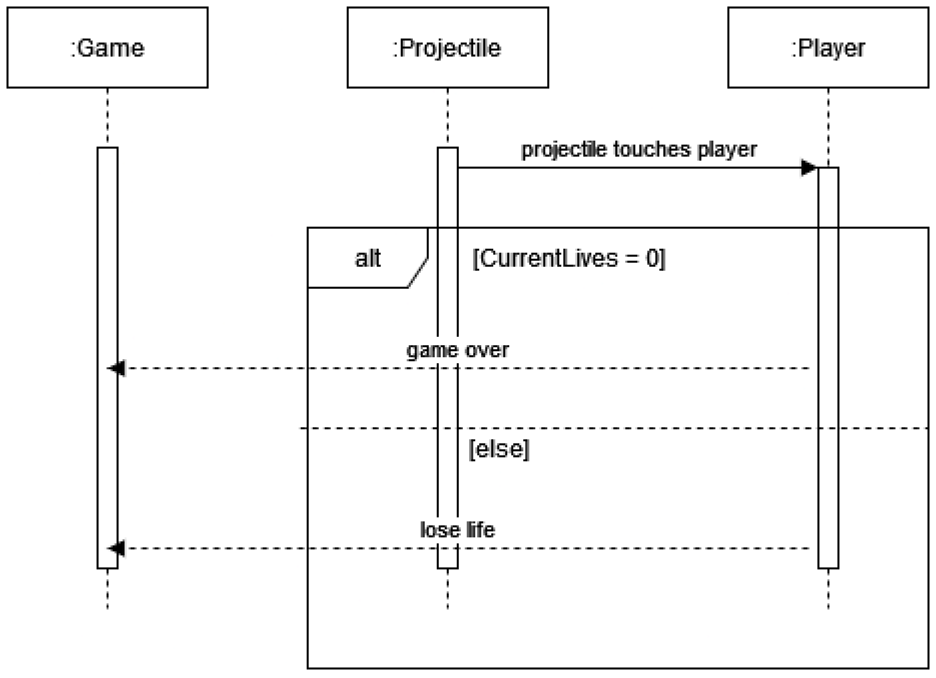
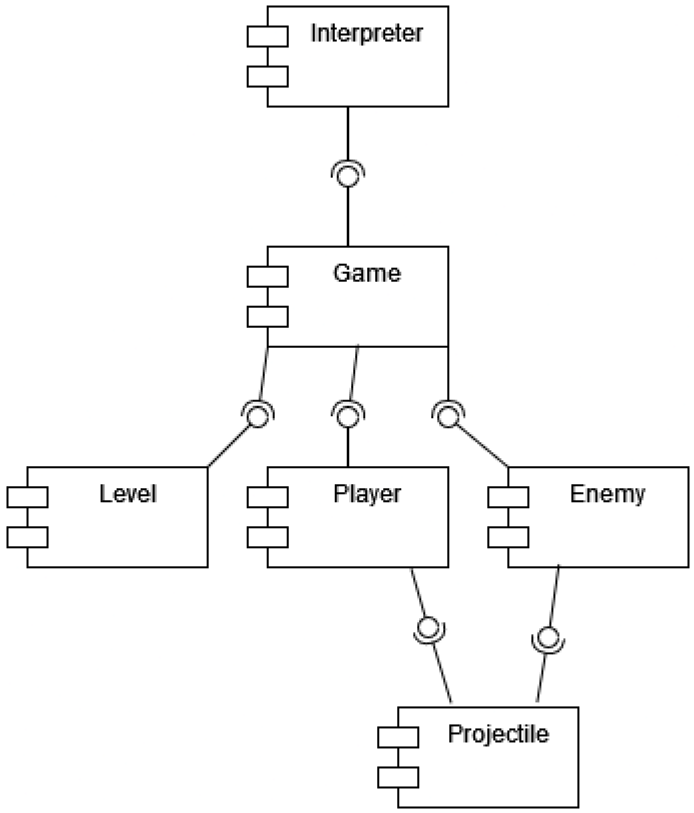


Figure 2-3: Sequence diagram for player getting hit by projectile



2.1 Overview

Figure 2-4: Software architecture component diagram

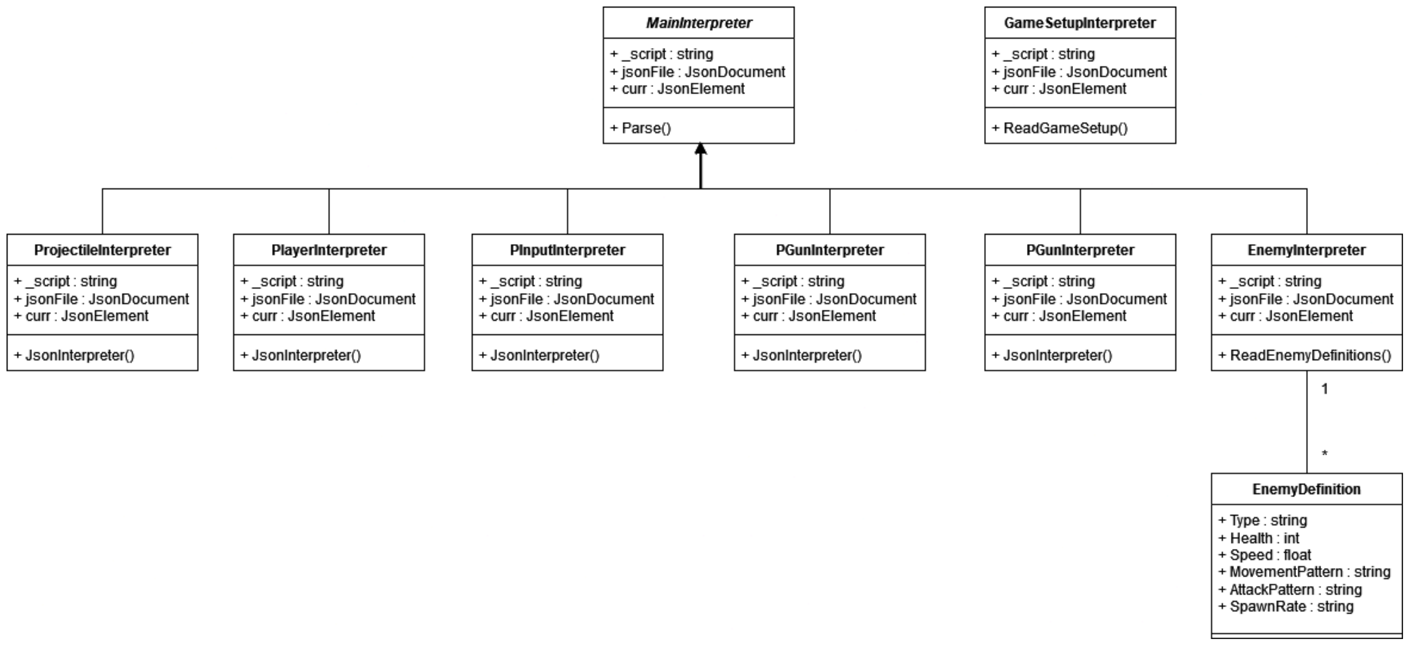


Our overall software design follows a Multi-layered architectural pattern, where the Game component provides the foundation of the user-facing experience. The Interpreter component requires all three of the Game sub-components to function since all of the interpreter information is passed down to them to create the specific Game level environment. The Projectile subsystem is used by the Player and Enemy components.

2.2 Subsystem Decomposition

2.2.1 Interpreter Subsystem

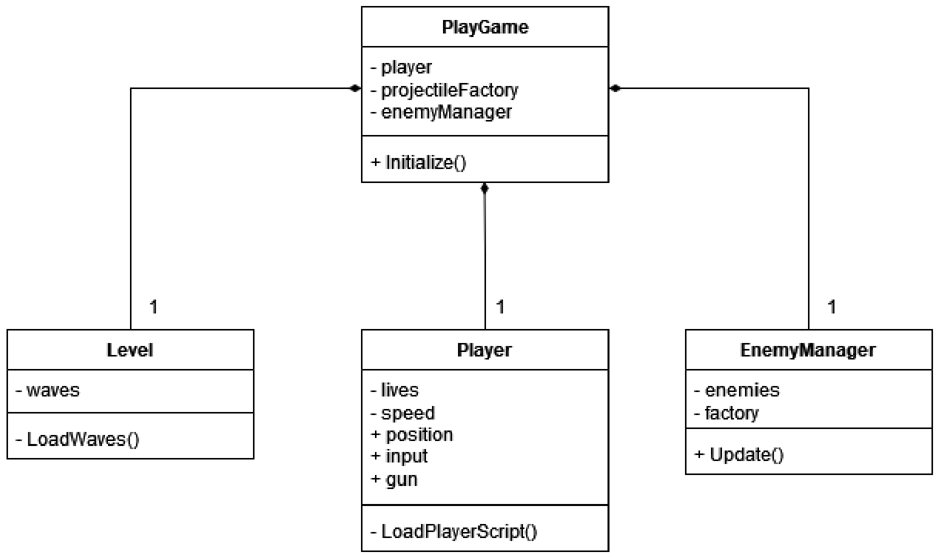
Figure 2-5: Class diagram for interpreter subsystem



The interpreters that inherit from the MainInterpreter abstract class create the basis for the game environment. The GameSetupInterpreter class deals with more aesthetic elements of the game environment, such as how the background looks, music that plays, sfx volume, etc.

2.2.2 Game Subsystem

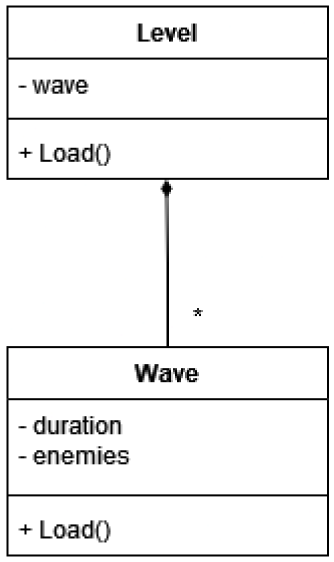
Figure 2-6: Class diagram for game subsystem



The Game subsystem consists of a main PlayGame class which functions as the user-facing game window, and this is where a lot of the MonoGame engine work comes in. Parts of the Level, Player, and Enemy subsystems are used as fields within this PlayGame environment object.

2.2.3 Level Subsystem

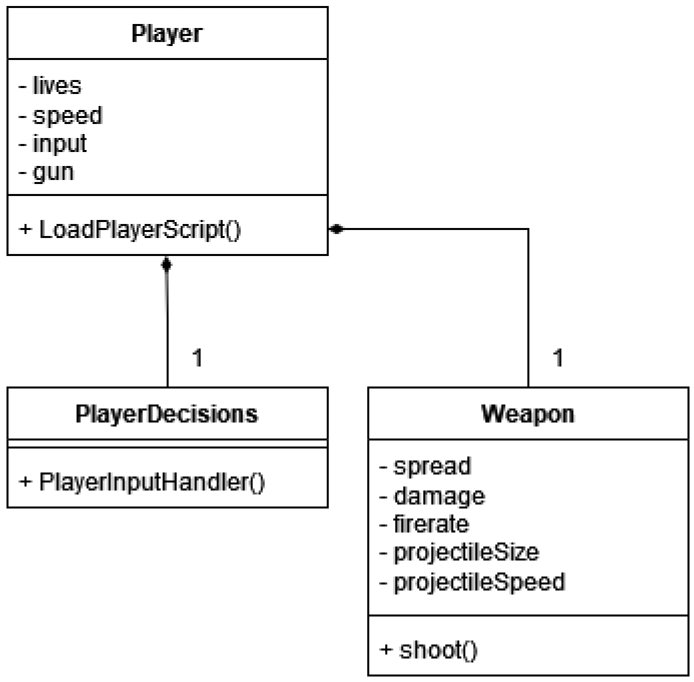
Figure 2-7: Class diagram for level subsystem



The Level subsystem is composed of a Level class which then contains an array of Wave components, and in each Wave component, there is info such as duration and what enemies to spawn during this wave.

2.2.4 Player Subsystem

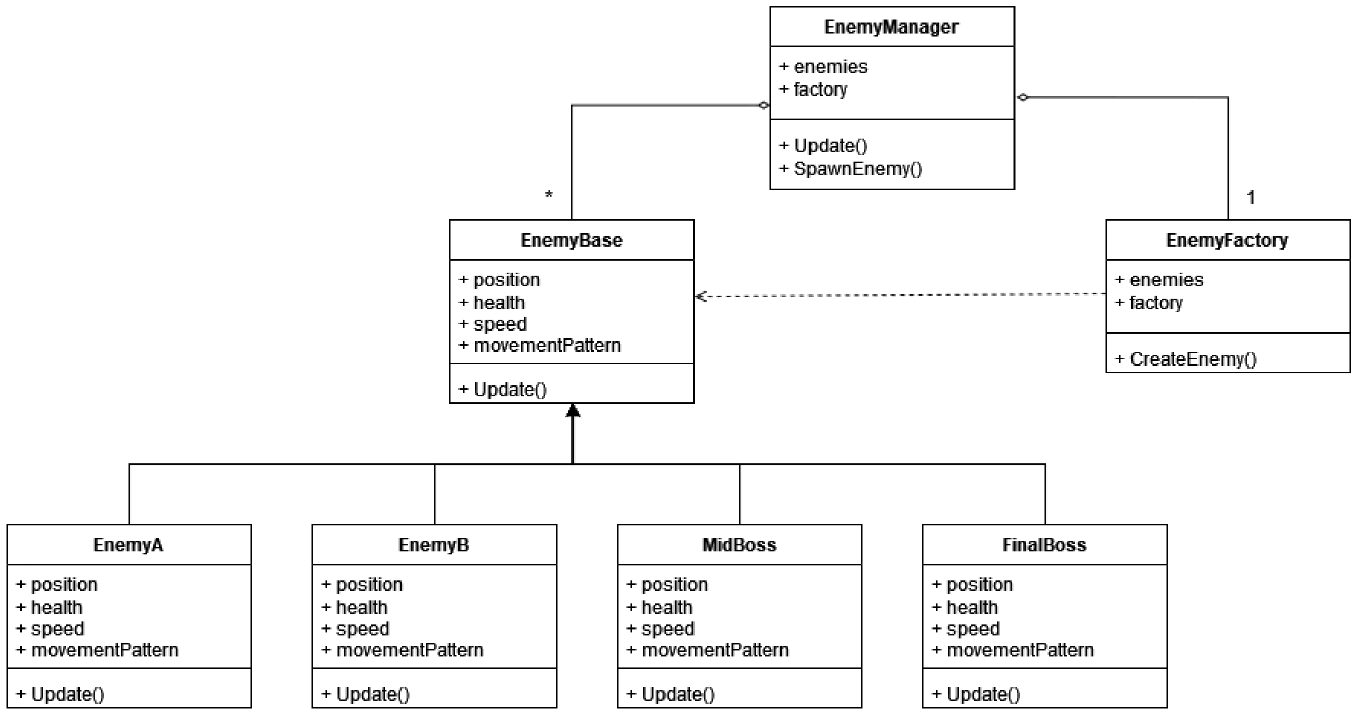
Figure 2-8: Class diagram for player subsystem



The Player subsystem includes the Player class which uses the PlayerDecisions class to handle user input via keys, and the Weapon class is the class that deals with shooting projectiles from the player's position on screen. Player lives and similar stats are also stored in the main Player class.

2.2.5 Enemy Subsystem

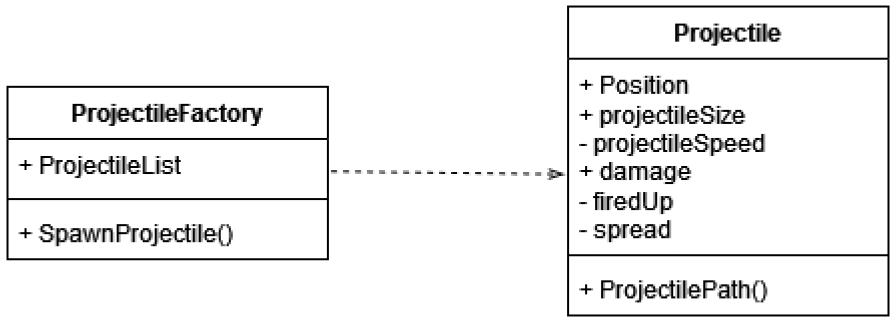
Figure 2-9: Class diagram for enemy subsystem



The Enemy subsystem consists of a factory that makes each subtype of enemy, with an EnemyManager class built to hold an array of enemies created by the factory. The manager handles enemies in the game environment, updating every game tick in order to remove any dead enemies or spawn new ones.

2.2.6 Projectile Subsystem

Figure 2-10: Class diagram for projectile subsystem



The projectiles are created by the ProjectileFactory, and contain damage and firedUp fields for player projectiles.

2.2.7 Design Patterns

Three design patterns used in our software design include the Factory, Builder, and Command design patterns.

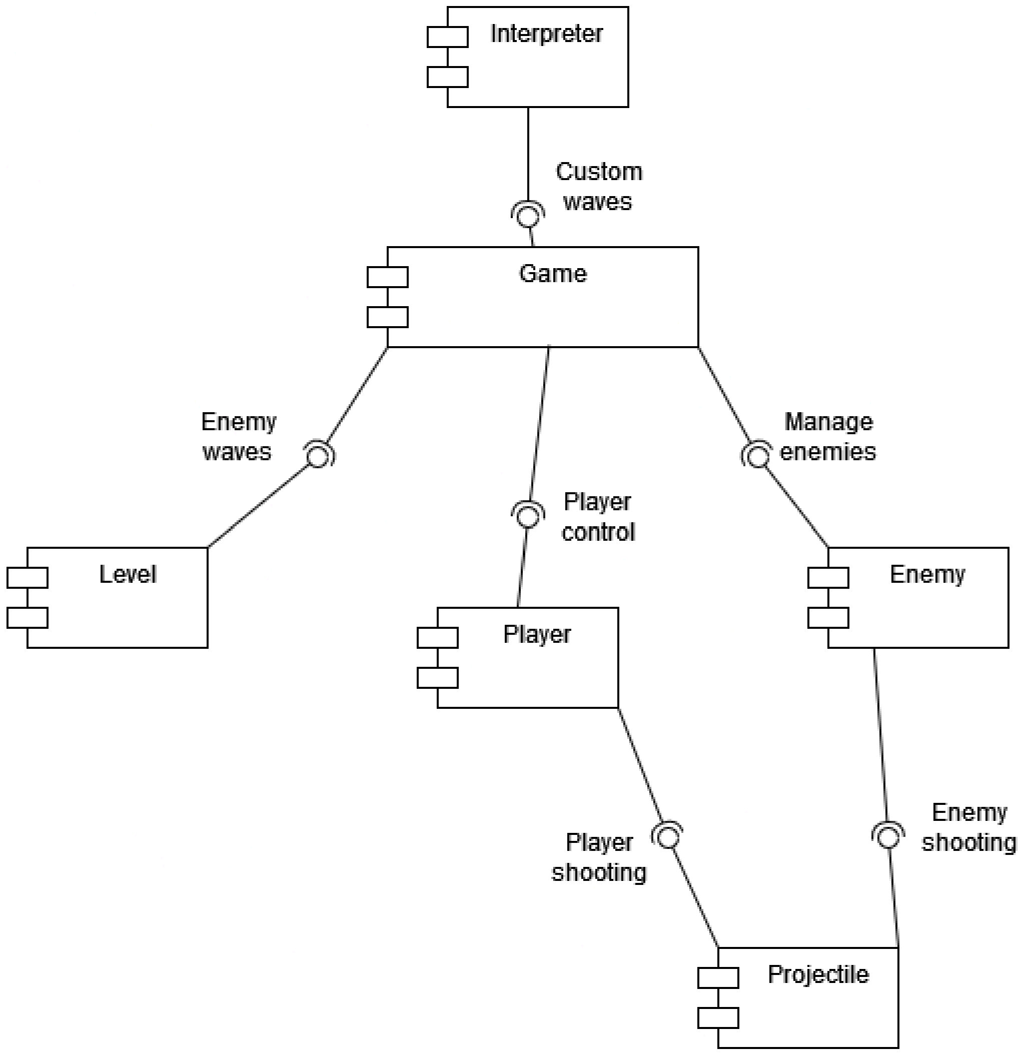
Factory: The Enemy and Projectile classes use the Factory design pattern to create enemies or projectiles of a certain type.

Builder: The level interpreter uses the Builder design pattern to build the desired level. The JSON files are parsed and put together piece-by-piece in the game logic to create the fully functioning game environment.

Command: The EnemyManager class is an example of the Command design pattern being used in our project. It has control of all of the enemies currently alive in the game environment, and is able to delete enemies which have lost all of their hit points and died.

3. Subsystem Services

Figure 3-1: Subsystem service component diagram



Dependencies:

Enemy subsystem is responsible for all enemy logic and uses the Projectile subsystem to get enemies to shoot projectiles.

Player subsystem is responsible for all player logic and uses the Projectile subsystem to get player to shoot projectiles using the player's Weapon class.

Level subsystem is responsible for enemy wave logic and uses the Enemy subsystem from within the Game subsystem to spawn enemies.

The Game subsystem is responsible for the user-facing game environment and uses the Player, Enemy, Level, and Projectile subsystems to provide game functionality.

The Interpreter subsystem is responsible for loading custom levels and requires the Game subsystem to load the level properly.

Services:

Custom waves: Provided by Game subsystem, required by Interpreter subsystem in order to built the custom levels.

Enemy waves: Provided by Level subsystem, required by Game subsystem to spawn enemies in waves.

Player control: Provided by Player subsystem, required by Game subsystem to control the player character in the game environment.

Manage enemies: Provided by Enemy subsystem, required by Game subsystem to manage enemies attacking and dying properly.

Player shooting: Provided by Projectile subsystem, required by Player subsystem to shoot projectiles that damage enemies.

Enemy shooting: Provided by Projectile subsystem, required by Enemy subsystem to shoot projectiles to kill the player.