Software Design Document

for

ShakeAndBake

Team: ShakeAndBake

Project: ShakeAndBake

Team Members:
Cole Bennett
Austin Marino
Ben Hamlin
Sam Schreiber

Last Updated: [3/10/2019 5:05:00 PM]

Table of Contents

Document Revision History

Revision Number	Revision Date	Description	Rationale
1.0	March 10, 2019	Deliverable 2 Document	Initial version of our design document which includes an introduction to the project, an overview of the software architecture, and a description of the subsystem services. UML diagrams for each section are also included in this version.

List of Figures

1. Introduction

Our game, ShakeAndBake, is a bullet hell game, a subgenre of shooters, where the main gameplay focus is dodging seemingly endless waves of different types of projectiles. These projectiles sometimes form certain patterns to increase the difficulty and/or aesthetics of the game. In our game the player controls a character which flies around the screen, shooting at waves of enemies until eventually reaching a stage boss. At the end of each stage (or sometimes in the middle), the player will fight the stage boss who is usually more powerful and have more elaborated attacks. During the extent of the game the user can achieve power-ups giving them radom special abilities to better protect themselves and/or attack the enemies. To win the game, the use must survive all the stages, including the final boss round, without losing all their lives. The end goal for our project is to design and implement a stand-alone desktop game and its corresponding level interpreter. In the implementation of this game programatically we are extensively using design patterns and a clean software architecture to ensure our code is as modular, extendable and efficient as possible. Ultimately, the purpose of this project is to provide a platform on which we can exercise software design and decision making on software architecture through the creation of our game.

1.1 Architectural Design Goals

Aside from our overall design goal of having a system comprised of modular and loosely-coupled subsystems, we have several design qualities that we focus on which we believe enhance the overall quality and user experience of our application. These design goals, specified below, are aimed at ensuring our system is free from deficiencies or defects and contain all the necessary attributes needed to ensure our runtime behavior, system design, and user experience are streamlined, efficient and clean.

1.1.1 Design Quality 1

The primary objective that our system focuses on is performance. The most intensive operations in our game's update loop reside in the Collision Board component. Due to the large number of Game Objects which may be active in our game at a given time, we chose to introduce this component to place an emphasis on efficient collision detection. This component utilizes spatial partitioning by dividing our board into buckets which each maintain a collection of active Game Objects that are in the bounds of a given bucket. This allows us to drastically improve the lookup time in our collision detection algorithm as we only have to search neighboring buckets rather than iterating through a possibly large collection of Game Objects multiple times for each check. One of the main guiding factors in our decision to implement collision detection in this way was to ensure that the user would never experience any lagging due to intense and inefficient game logic, even as we add more features in future milestones.

1.1.2 Design Quality 2

The second design quality we focus on in our application is modifiability and scalability. Most games nowadays release new DLCs (additional content created for a released video game) regularly in an attempt to keep users engaged and excited to play their game. With this in mind we have designed our game engine to avoid hard coded settings and instead read gameplay details and logic from external JSON files. The engine then interprets the file to dictate the AI behaviors, such as enemies and projectiles, throughout different stages of the game. This allows us to continually change the gameplay experience for our user without needing to actually change any of our existing code. An additional benefit includes the ability to swap the game configuration at any time, acting as a modular component to our application. In terms of scalability, we used the MVC architecture in an attempt to separate the three essential subsystems our our program, Model, View, and Controller. By doing so we were able to reduce coupling across our system while at the same time increasing cohesion within each subsystem. With this architectural design pattern in place we are able to protect any future implementation changes from triggering a domino effect, in which a change made to one class, method, etc. can affect others in strange and often unintended ways.

2. Software Architecture

Below are UML sequence diagrams representing three key use cases for our game, which include Player Shoots Projectile, Enemy Spawns, and Player Wins Game.

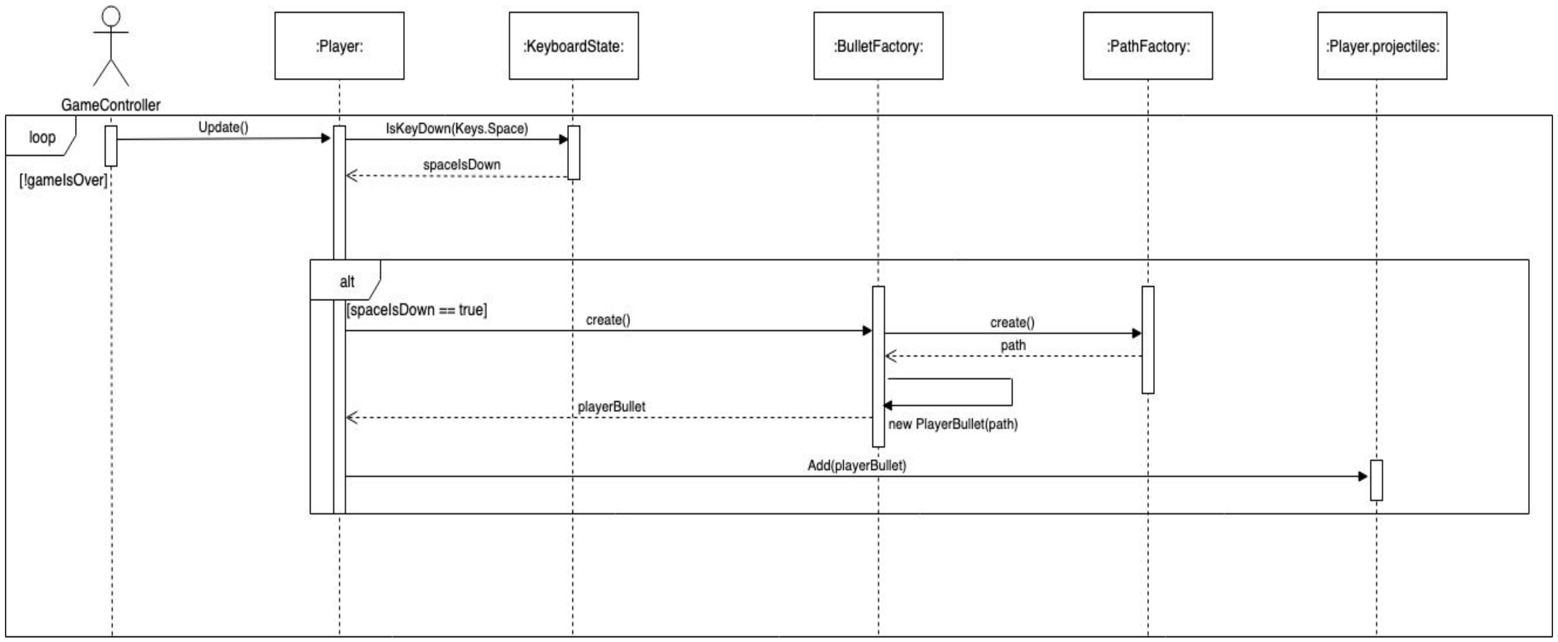

Figure 1-1. Player Shoots Projectile - Sequence Diagram

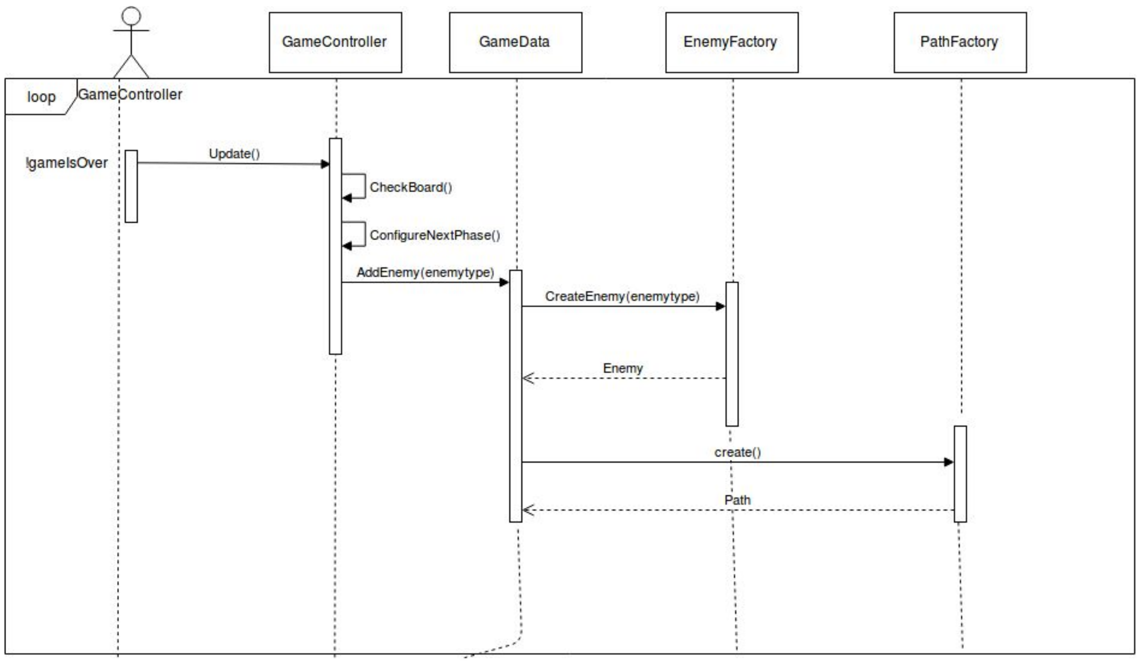

Figure 1-2. Enemy Spawns - Sequence Diagram

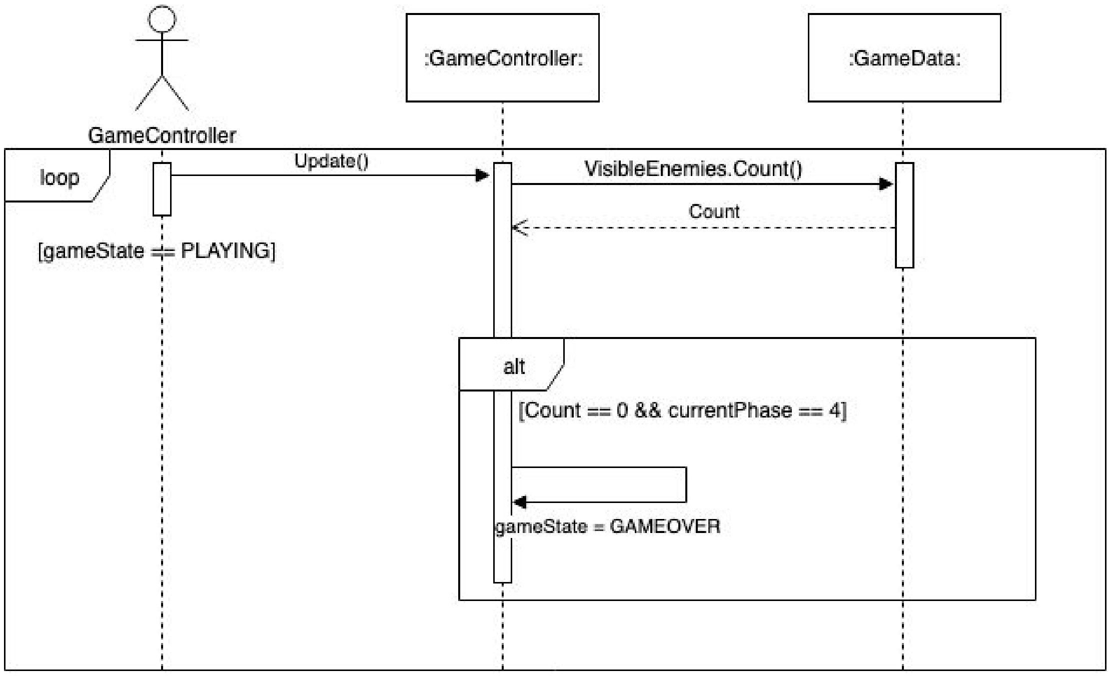

Figure 1-3. Player Wins Game - Sequence Diagram

2.1 Overview

Our design model follows the Model-View-Controller (MVC) architectural pattern. We chose this pattern because of its ability to separate our application into three main logical components: the model, the view, and the controller. In our application, these components are represented by Game Data (Model), Game Board (View), and Game Controller (Controller). Each of these components are built to handle specific development aspects of our application, allowing us to write better organized, and therefore more maintainable code.

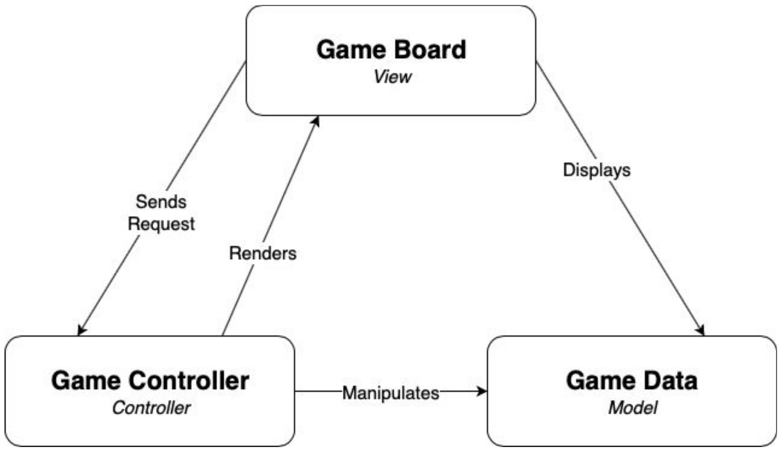

Figure 2-1. MVC Architecture - Component Diagram

2.2 Subsystem Decomposition

We decomposed our system into subsystems by utilizing the MVC architectural pattern and using our Game Controller as the Controller, our Game Board as our View, all data needed to run the game in our Game Data subsystem as the Model, and surfacing the necessary interfaces so that these subsystems may interact with one another. This allows us to separate the way we render our game, store our data, and manipulate our data so that our overall system is more modular, more cohesive, and our subsystems can scale independently. We will elaborate the responsibilities of each subsystem in the following sections.

2.2.1 Game Data Subsystem

The Game Data subsystem is in charge of storing and organizing all of the data needed by the Game Board and Game Controller subsystems. The data includes all Game Objects such as Enemies, Projectiles, and the Player. In addition, it includes the implementations of their respective creational and structural design patterns. It also exposes interfaces which perform operations related to fetching and querying data while abstracting as much detail about how those operations are performed as possible. An example of a query operation supported by the Game Data subsystem is determining whether a Game Object has collided with another Game Object by utilizing the Collision Board component which uses spatial partitioning for collision detection. Lastly, our Game Data subsystem also contains all the data-related logic that the user works with and other business logic/data that only the model itself can use. It has no knowledge about the other two subsystems and thus only takes orders and doesn't give any itself.

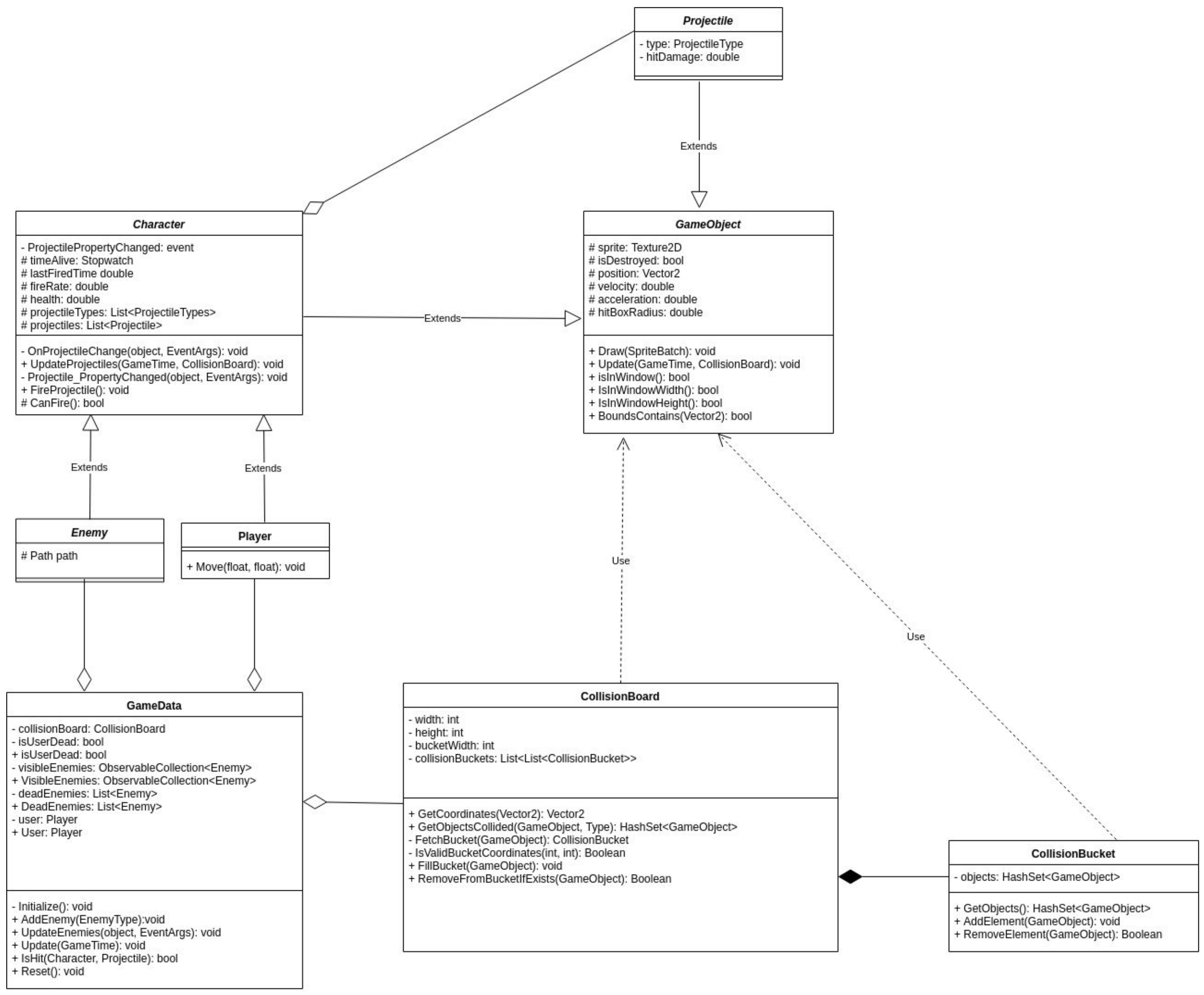
Projectile
- type: ProjectileType
- hitDamage: double
Extends
Character
- ProjectilePropertyChanged: event
# timeAlive: Stopwatch
# lastFiredTime double
# fireRate: double
# health: double
# projectileTypes: List<ProjectileTypes>
# projectiles: List<Projectile>
- OnProjectileChange(object, EventArgs): void
+ UpdateProjectiles(GameTime, CollisionBoard): void
- Projectile_PropertyChanged(object, EventArgs): void
+ FireProjectile(): void
# CanFire(): bool
Extends
GameObject
# sprite: Texture2D
# isDestroyed: bool
# position: Vector2
# velocity: double
# acceleration: double
# hitBoxRadius: double
+ Draw(SpriteBatch): void
+ Update(GameTime, CollisionBoard): void
+ isInWindow(): bool
+ IsInWindowWidth(): bool
+ IsInWindowHeight(): bool
+ BoundsContains(Vector2): bool
Extends
Extends
Enemy
# Path path
Player
+ Move(float, float): void
Use
Use
GameData
- collisionBoard: CollisionBoard
- isUserDead: bool
+ isUserDead: bool
- visibleEnemies: ObservableCollection<Enemy>
+ VisibleEnemies: ObservableCollection<Enemy>
- deadEnemies: List<Enemy>
+ DeadEnemies: List<Enemy>
- user: Player
+ User: Player
- Initialize(): void
+ AddEnemy(EnemyType):void
+ UpdateEnemies(object, EventArgs): void
+ Update(GameTime): void
+ IsHit(Character, Projectile): bool
+ Reset(): void
CollisionBoard
- width: int
- height: int
- bucketWidth: int
- collisionBuckets: List<List<CollisionBucket>>
+ GetCoordinates(Vector2): Vector2
+ GetObjectsCollided(GameObject, Type): HashSet<GameObject>
- FetchBucket(GameObject): CollisionBucket
- IsValidBucketCoordinates(int, int): Boolean
+ FillBucket(GameObject): void
+ RemoveFromBucketIfExists(GameObject): Boolean
CollisionBucket
- objects: HashSet<GameObject>
+ GetObjects(): HashSet<GameObject>
+ AddElement(GameObject): void
+ RemoveElement(GameObject): Boolean

Figure 2-2. Game Data Subsystem

2.2.2 Game Board Subsystem

The Game Board subsystem serves as the user interface and is what is displayed to the users and how they interact with the game. It simply retrieves and displays the data contained in the Game Data subsystem, and enables the user to modify said data.

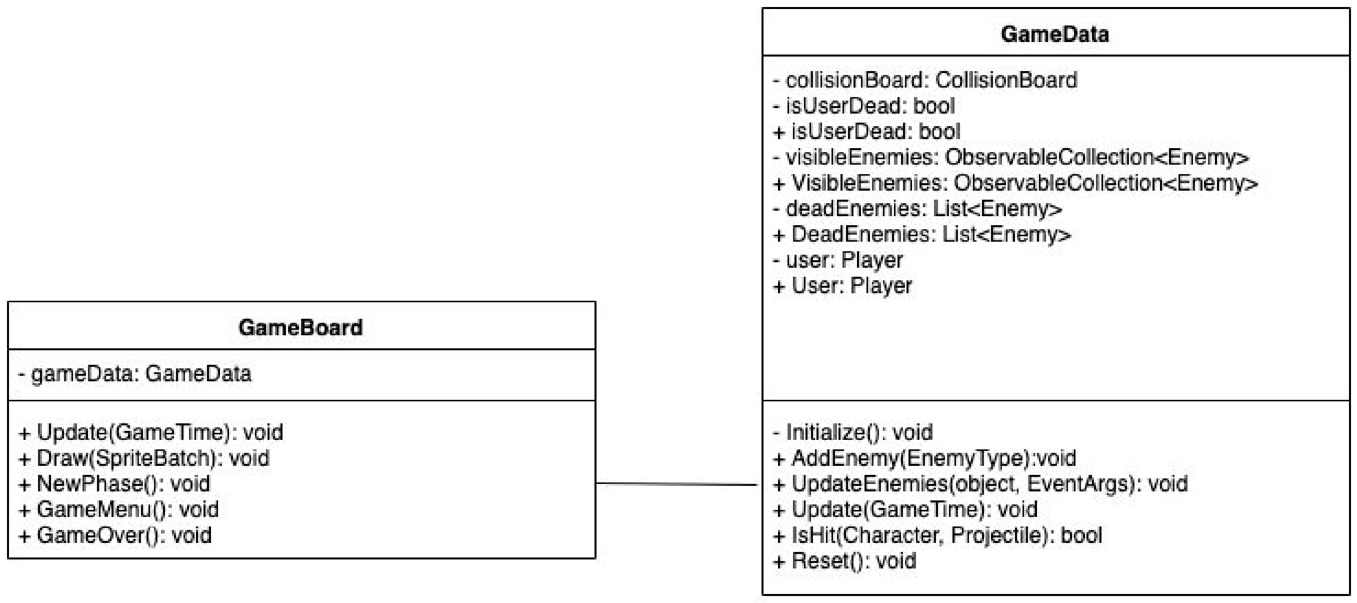
GameBoard
- gameData: GameData
+ Update(GameTime): void
+ Draw(SpriteBatch): void
+ NewPhase(): void
+ GameMenu(): void
+ GameOver(): void
GameData
- collisionBoard: CollisionBoard
- isUserDead: bool
+ isUserDead: bool
- visibleEnemies: ObservableCollection<Enemy>
+ VisibleEnemies: ObservableCollection<Enemy>
- deadEnemies: List<Enemy>
+ DeadEnemies: List<Enemy>
- user: Player
+ User: Player
- Initialize(): void
+ AddEnemy(EnemyType):void
+ UpdateEnemies(object, EventArgs): void
+ Update(GameTime): void
+ IsHit(Character, Projectile): bool
+ Reset(): void

Figure 2-3. Game Board Subsystem

2.2.3 Game Controller Subsystem

The Game Controller subsystem handles user requests. It is the decision maker and the glue between the the Game Data and Game Board subsystems. Our controllers main job is to tell the Game Data subsystem to update itself, providing it with the necessary parameters to do so, and then informs the Game Board to update its display to reflect the newest changes. Ultimately, it serves as an interface between the Game Data and Game Board subsystems to process all the business logic and incoming requests, manipulate data using the GameData subsystem and interact with the Game Board to render the final output.

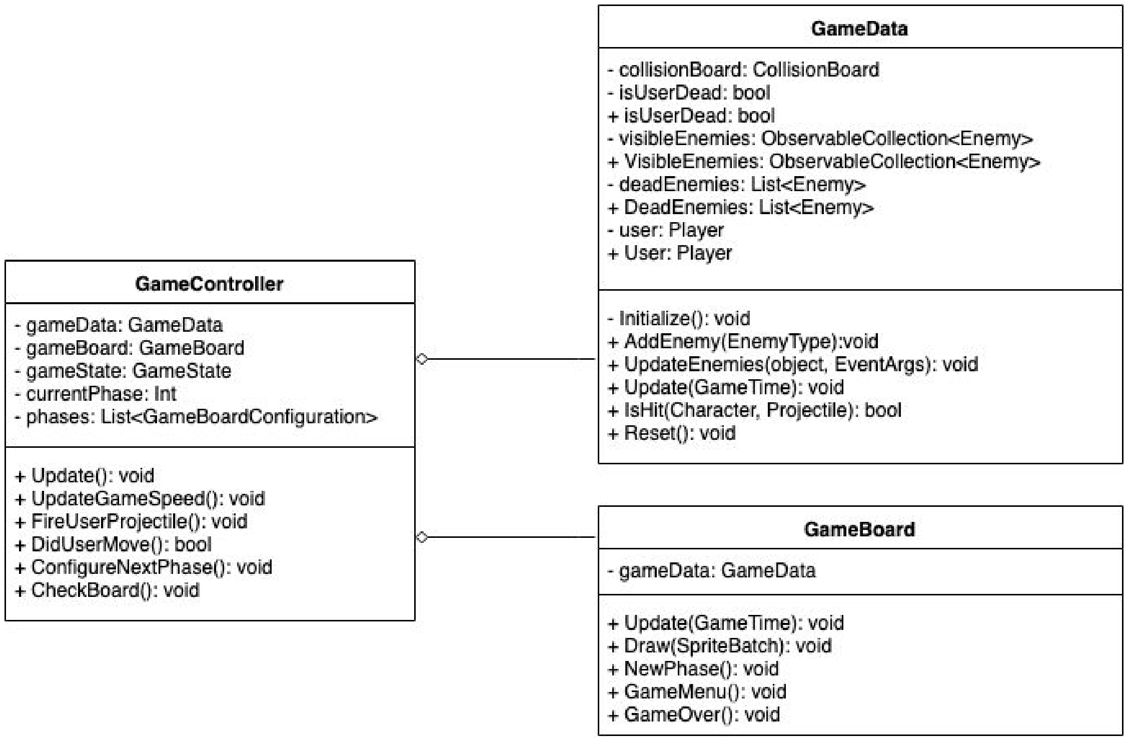
GameData
- collisionBoard: CollisionBoard
- isUserDead: bool
+ isUserDead: bool
- visibleEnemies: ObservableCollection<Enemy>
+ VisibleEnemies: ObservableCollection<Enemy>
- deadEnemies: List<Enemy>
+ DeadEnemies: List<Enemy>
- user: Player
+ User: Player
- Initialize(): void
+ AddEnemy(EnemyType):void
+ UpdateEnemies(object, EventArgs): void
+ Update(GameTime): void
+ IsHit(Character, Projectile): bool
+ Reset(): void
GameController
- gameData: GameData
- gameBoard: GameBoard
- gameState: GameState
- currentPhase: Int
- phases: List<GameBoardConfiguration>
+ Update(): void
+ UpdateGameSpeed(): void
+ FireUserProjectile(): void
+ DidUserMove(): bool
+ ConfigureNextPhase(): void
+ CheckBoard(): void
GameBoard
- gameData: GameData
+ Update(GameTime): void
+ Draw(SpriteBatch): void
+ NewPhase(): void
+ GameMenu(): void
+ GameOver(): void

Figure 2-4. Game Controller Subsystem

2.2.4 Design Patterns

This section includes three examples of design patterns used in our application. Although we have multiple implementations of factories, we chose to include a UML diagram representing our Enemy abstract factory as it is the most complex case.

2.2.4.1 Abstract Factory for Enemies

Abstract Factory creational pattern fits well with creating enemies since we have multiple types which all inherit from the Enemy base abstract class. The EnemyFactory abstract class is our Abstract Factory and it contains a single method Create() which returns an Enemy instance. Enemy is our Abstract Product. EasyEnemy, MediumEnemy, HardEnemy, MidBoss, and FinalBoss are our Concrete Products. Corresponding to each Concrete Product, we have a Concrete Factory.

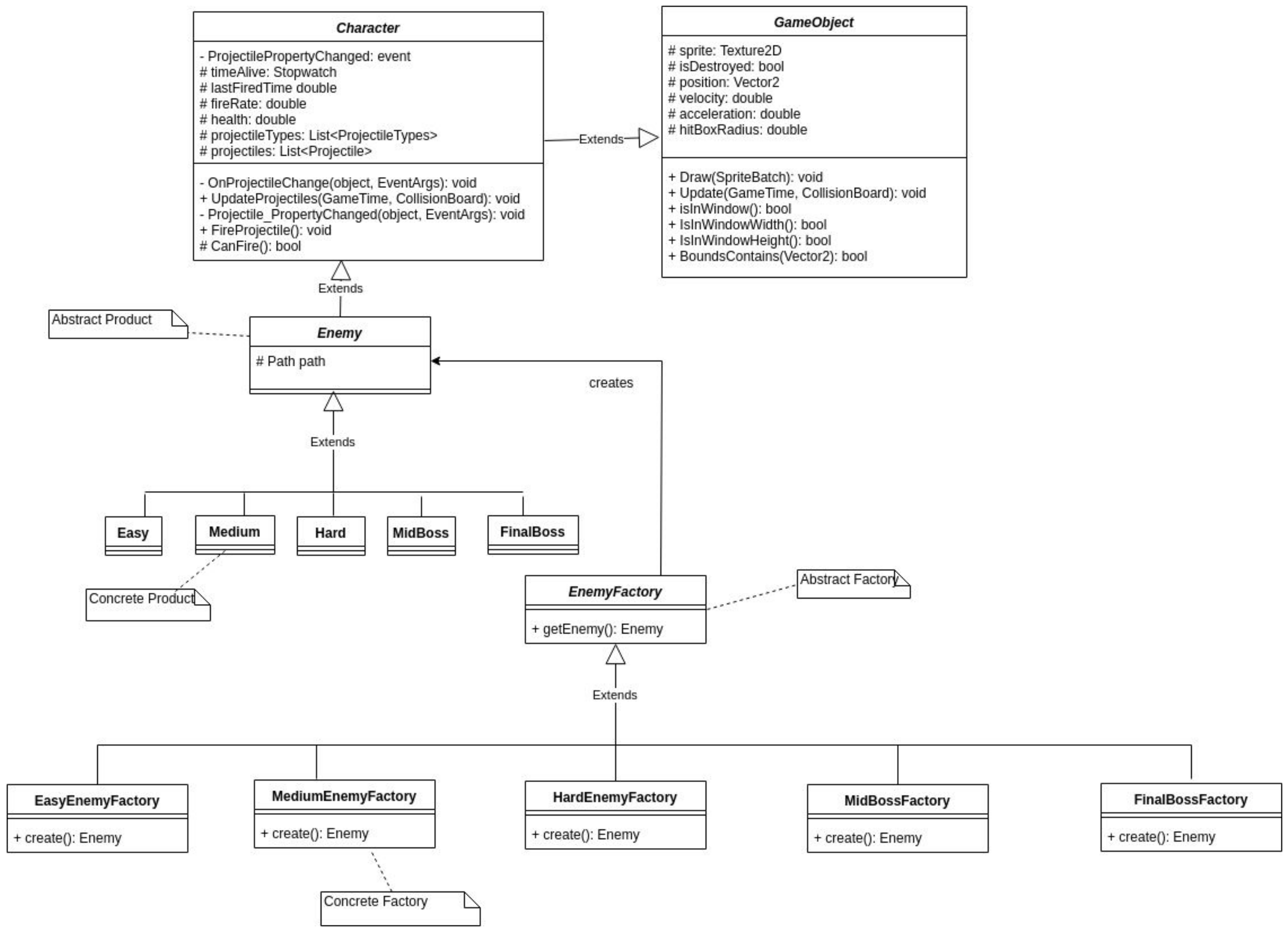

Figure 2-5. Abstract Factory for Enemy

2.2.4.2 Composite Pattern for Game Objects

The root of all entities in our Game Data subsystem is the GameObject abstract class. We found that the Composite structural pattern fits well with our application as we use GameObject as the Component, and other derived classes as our Composite and Leaf nodes. Player, which inherits from Character, is a Leaf node. Enemy, which also inherits from Character, is a Composite node as it contains a list of child Components i.e. Composite or Leaf nodes) to introduce additional features for each enemy type.

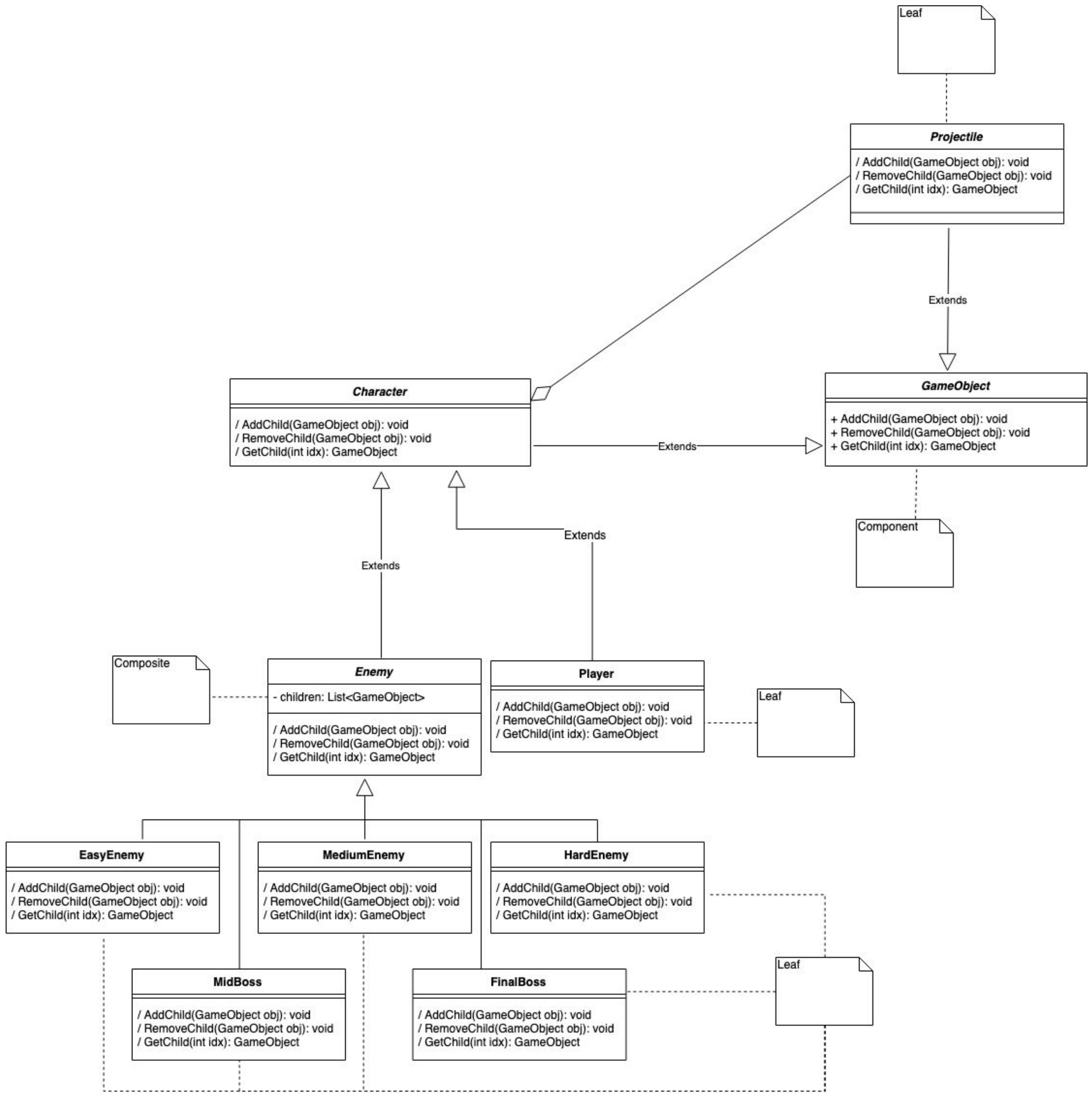
Leaf
Projectile
/ AddChild(GameObject obj): void
/ RemoveChild(GameObject obj): void
/ GetChild(int idx): GameObject
Extends
Character
/ AddChild(GameObject obj): void
/ RemoveChild(GameObject obj): void
/ GetChild(int idx): GameObject
Extends
GameObject
+ AddChild(GameObject obj): void
+ RemoveChild(GameObject obj): void
+ GetChild(int idx): GameObject
Component
Extends
Extends
Composite
Enemy
- children: List<GameObject>
/ AddChild(GameObject obj): void
/ RemoveChild(GameObject obj): void
/ GetChild(int idx): GameObject
Player
/ AddChild(GameObject obj): void
/ RemoveChild(GameObject obj): void
/ GetChild(int idx): GameObject
Leaf
EasyEnemy
MediumEnemy
HardEnemy
MidBoss
FinalBoss
/ AddChild(GameObject obj): void
/ RemoveChild(GameObject obj): void
/ GetChild(int idx): GameObject
Leaf

Figure 2-6. Composite Pattern for Game Objects

2.2.4.3 Singleton Pattern for Player

Since we only need one Player instance, we utilize the Singleton creational pattern to enforce this desired constraint by making the Player constructor private, storing a private static Player instance in the class, and adding a public static GetInstance() method. Also, we are using lazy initialization for setting the Player instance. Note that our UML diagram only contains the information related to our use of the Singleton pattern.

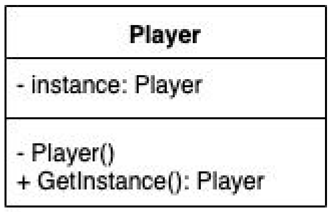

Figure 2-7. Singleton Pattern for Player

3. Subsystem Services

Our application follows the Model-View-Controller architectural pattern. Subsequently, the use of this pattern introduces three subsystems described in **Section 2.2**, which include Game Data (Model), Game Board (View), and Game Controller (Controller). There are several dependencies among these three subsystems. Therefore, we have multiple services which have a corresponding interface to facilitate interactions between these subsystems.

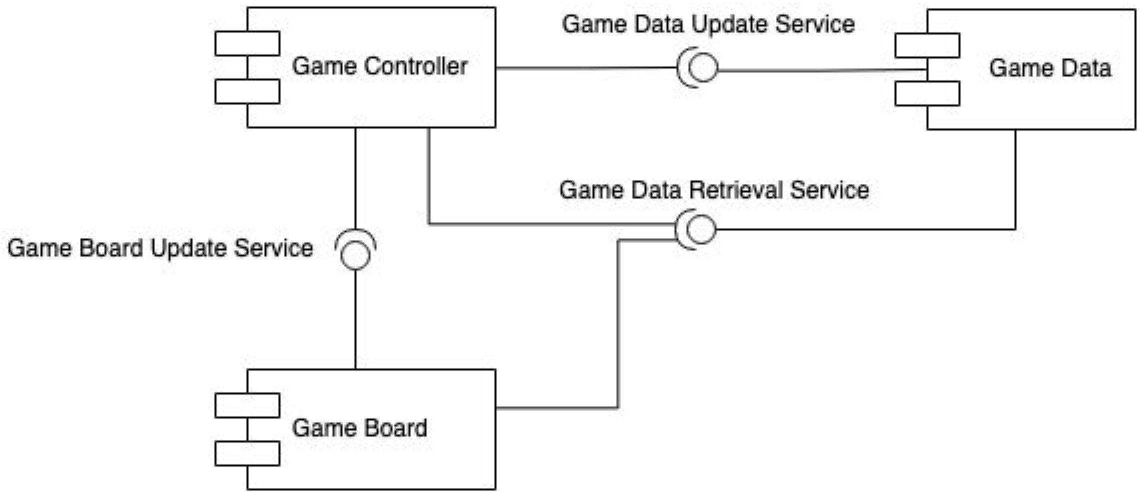

Figure 3-1. Subsystem Services - Component Diagram

3.1. Game Board Update Service

The first dependency is between the Game Controller subsystem and the Game Board subsystem. With regards to this service, the Game Controller is responsible for initiating all updates to the Game Board. The main function in the interface of this system is GameBoard.Update(). The Game Controller calls this function to initiate updates on the Game Board, which is our view. In addition, the Game Board requires the Game Controller to call this function in the service interface for the view to update and render.

3.2. Game Data Retrieval Service

This service is utilized by both the Game Controller and Game Board subsystems. The Game Controller is responsible for updating the active Player, but to do so it must first retrieve the Player by calling

GameData.User(). The Game Board calls GameData.VisibleEnemies() to retrieve the collection of active enemies in the game to render. This service is required by both of these subsystems as the data must be retrieved first before any updates can be made on our Game Objects.

3.3. Game Data Update Service

This service is utilized by the Game Controller subsystem. The Game Controller is responsible for initiating updates to all Game Objects, which are managed by our Game Data subsystem. This service has one main function in the service interface which is GameData.Update(). Other operations include GameData.AddEnemy() and direct calls which modify Game Objects such as GameData.User.FireProjectile(). Game Data requires the Game Controller to call these operations in the service interface for the Game Objects to be updated.