Software Design Document for

Team: Code Crusaders
Project: Road Rage
Team Members:
Connor Strom
Emma Fletcher
Logan Kloft
Manjesh Puram

Last Updated: 3/26/2023 4:44 PM

Table of Contents

Document Revision History

Revision Number	Revision Date	Description	Rationale
1.0	3/26	Submission for gate review	The first document submitted for review

List of Figures

List of Tables

1. Introduction

Road Rage is a bullet hell game that derives some of its functionality from the popular Touhou Project game series. Although Road Rage is a game, its emphasis is to practice software design patterns and architectures. Therefore, the game is designed to be as robust and as flexible as possible. For information about how Road Rage as a game is supposed to look, please read the “Project Vision” document. The rest of this document focuses on the design of Road Rage starting with its goals, then venturing into subsystem design and finally ending with service decomposition.

1.1 Architectural Design Goals

Road Rage’s system has a focus on flexibility, extensibility, configurability, and performance. The following outline in better detail the design goals that impact Road Rage’s system architecture:

The game should:

- Provide flexibility for interchangeable enemy behavior

If the level creator decides they want to change the movement behavior of an enemy for example, it should require only a simple change made in one spot not multiple, and should not require the user to consider impacts to the internal state.

- Provide flexibility for interchangeable shot behavior

If the level creator decides they want to change the pattern of shots of an enemy at a specific time and for a specific duration, they should be able to make the change without worrying about how it will affect the internal state of the program.

- Provide flexibility for interchangeable player behavior

Similar to the previous non-functional requirements, the level creator might decide they want to increase the speed of the player or change the player’s shot pattern; they should be able to do so without worrying about cascading impact effects within the program.

- Perform well on modern computers, e.g. without ‘frame drops’.

Consider the case that the level creator wishes to spawn many enemies at once, the design should be in such a way that spawning these enemies does not impact the movement of enemies, shots, or the player.

- Perform the same on two different modern computers

Two different computers will run at different speeds. The faster or slower computer should not cause a change in the speed of enemy, shot, and player behaviors. They should behave the same between the two computers.

- Allow configuration of enemy attributes

Factors that aren't a part of an enemies behavior such as its health, hitbox size, and sprite size should be adjustable by the level creator without requiring recompilation of the entire program.

- Allow configuration of shot attributes

Factors that aren't a part of a shot's behavior such as its damage, hitbox size, and sprite size should be adjustable by the level creator without requiring recompilation of the entire program.

- Allow configuration of player attributes

Factors that aren't a part of the player's behavior such the player's lives, hitbox size, and sprite size should be adjustable by the level creator without requiring recompilation of the entire program.

- Be extensible for developing new movement behaviors

On the development side of the architecture, creating new movement behaviors should require minimal change in the program and have little impact on the rest of the system.

- Be extensible for developing new shot behaviors

As with implementing new movement behaviors, creating new shot patterns should follow the same principles of minimal change and minimal impact. Adding a new feature should feel intuitive rather than like a treasure hunt with no treasure.

- Support adding surprise features as yet unrevealed

For a design to be truly flexible, it should support adding new features without having to rewrite large parts of a subsystem or subsystems.

The following design attributes support a flexible, extensible, configurable, and performant system architecture:

Performance

Performance is the ability to meet timing requirements and relates to scalability. In that case, that means satisfying the requirements that modern computers should not experience 'frame drops' or run differently. Performance in our architecture means managing the updating and creation of entities and sprites. There are two categories of tactics for managing performance. They are listed in addition to the tactics that we feel apply best to our scenario.

<u>Control Resource Demand Tactics:</u>

- *Limit event response*

Large events are 'blocking' because they run on the main thread, therefore we need to be careful about the impact of a single event which might cause a noticeable 'frame drop' if its impact takes too long to complete. For example, an event that triggers clearing a large amount of entities might take some time to remove the entities from a queue and therefore entities that aren't being cleared won't be rendered for a short duration of time, giving the perception that

they are not moving. We can limit event response by queuing intermediate states of long event-chains to be completed at a later time. This would mean designing events that don't immediately need to be handled and can wait for at least the next update cycle.

- *Prioritize events*

This relates to the previous point of deciding which events need to be handled and in which order. For our system, it makes sense to move entities first and immediately display the updated sprite location. This allows movement to happen not all at once so that it can be split up across multiple 'update' calls. After all entities finish moving and are drawn, hit detection can be applied before starting the process all over again. Prioritizing events in this way boosts the performance by allowing updates to occur in chunks rather than all at once which may cause stutters.

Manage Resources Tactics:

- *Introduce concurrency*

For update calls that don't impact other elements, such as the draw function, we can use threads to process the sprite array in equal parts. For update calls that do impact elements, such as potentially triggering a cascade of events caused by a collision detection, we can apply the same reasoning, but need a way to lock the cause of the event such that only one can be triggered at a time.

- *Bound queue sizes*

For computers who have different computing powers, they might only be able to process a limited amount of updates before causing noticeable 'frame drops'. This means we can define literal queue sizes where entities are stored in to be updated. Finding the right size that supports the best performance and widest range of modern computers would most likely not be a 'scientific' task but rather a 'what feels right'.

Testability

Testability refers to the ability to make software demonstrate its faults through testing. This can occur in different ways depending on the model of testing used. Best coverage occurs through the use of multiple models since one model cannot cover all test scenarios. The tactics used to allow for easier testing are split into two categories and the tactics we are exploring are explained as following:

Control and Observe System State Tactics:

- *Abstract Data Sources*

Using abstracted testing allows one to create tests more efficiently for objects with similar properties that need to be tested. An example of this is our Player, Shot, and Enemy entities which share the same abstract Entity class and also define similar function behaviors. We could test all three of these classes using an Entity object to make abstract testing interfaces.

- *Sandbox*

Sandboxing is isolating the system from the real world. We can accomplish this by making the player invulnerable to walk through an entire game to observe potential faults in the game loop or the display of entities.

Limit Complexity Tactics:

- *Limit structural complexity*

This refers to avoiding the dependence between components on each other. This makes testing easier because it means setting up a test doesn't require also setting up the cyclic dependency. In our case, we try to remove cyclic dependencies from subsystems by following the model-view-controller architectural design and within subsystems by having any one class not handle too many responsibilities.

2. Software Architecture

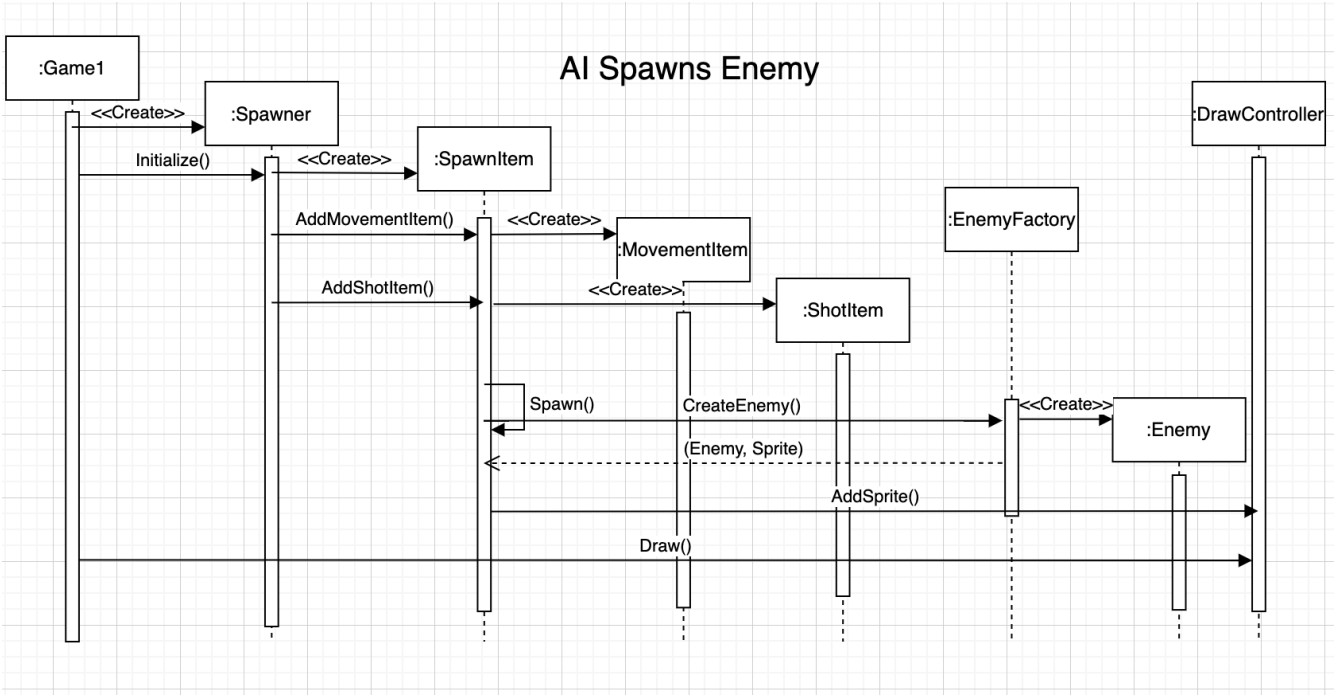

Figure 2.1: AI Spawns Enemy

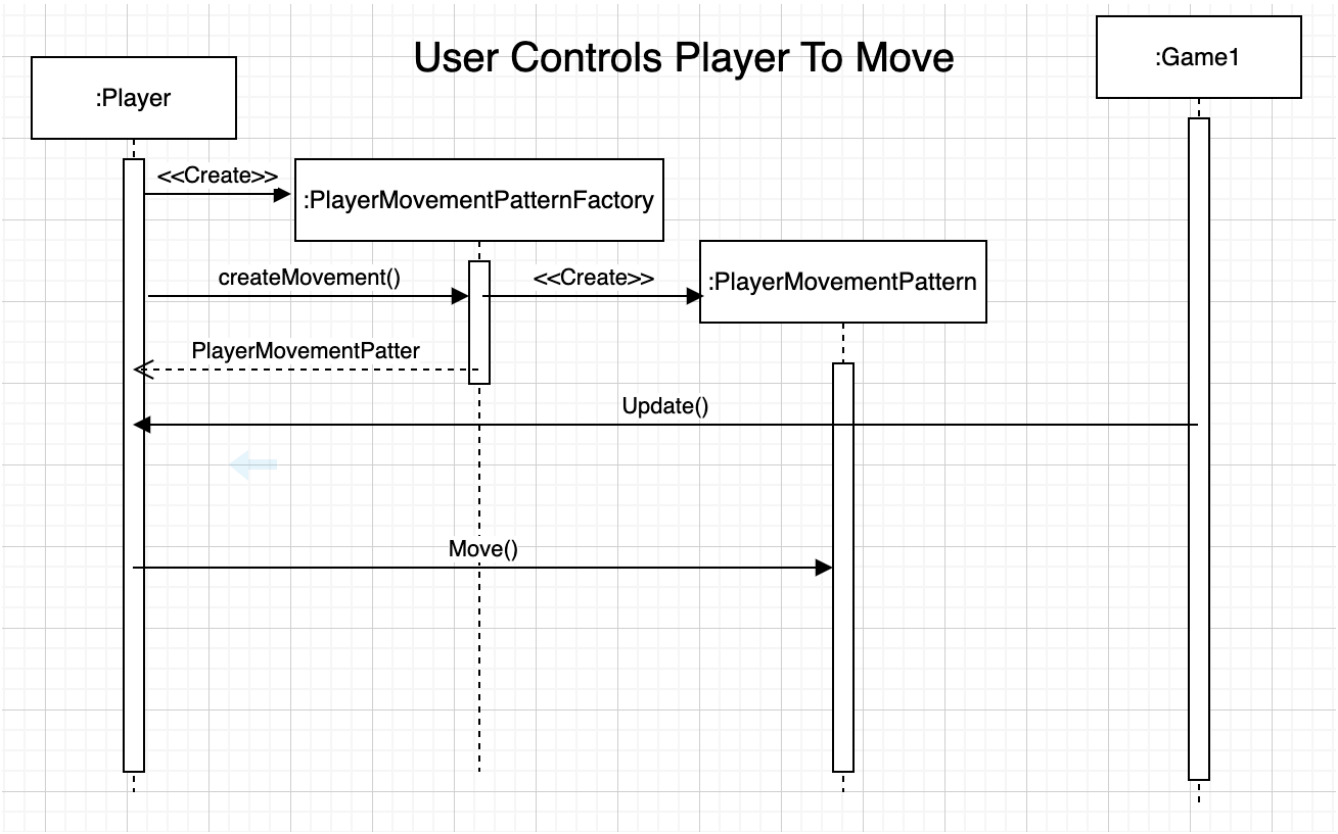

Figure 2.2: User Controls Player To Move

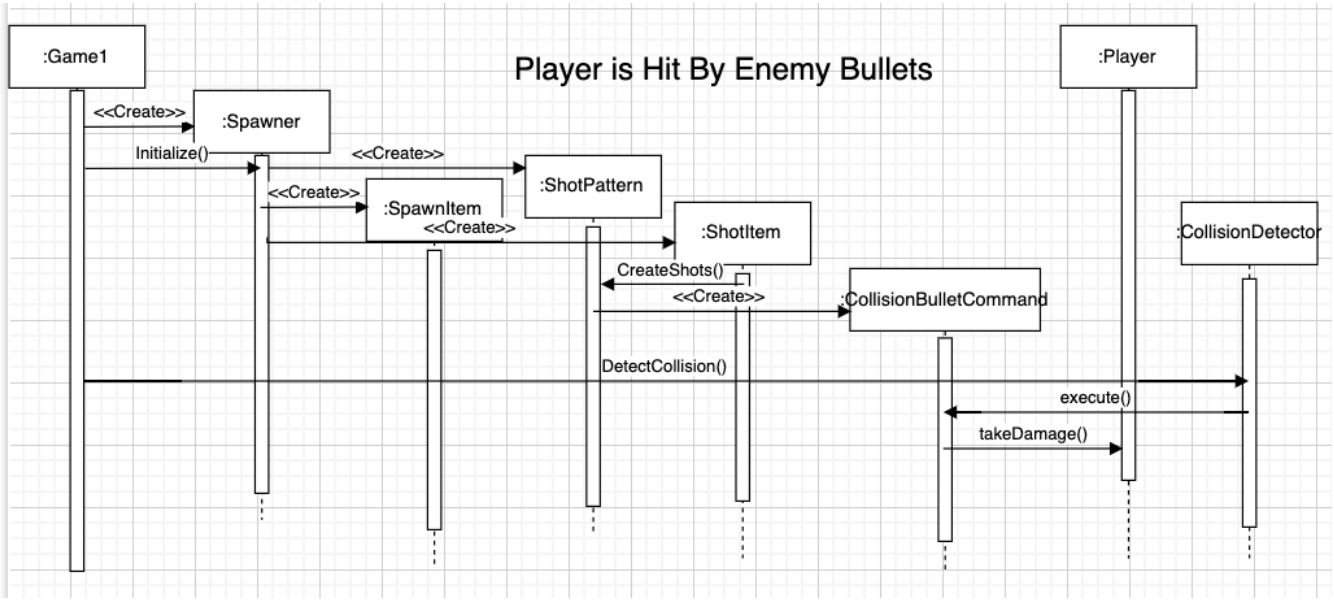

Figure 2.3: Player Is Hit By Enemy Bullets

2.1 Overview

Our aim for our project was to maintain a simple yet robust model to organize our features. Through this, we came down to using the Model-View-Controller or MVC for our design. We used this because it allowed us to separate our code into different sections which not only helped with organization but also with identifying where features need to be added and grouping related things together. In our program, we use all three of those components. In the Controller section, we keep all of our command, factory, movement, and shot patterns. In the Model section, we keep all of our entities which have their own separate features. Lastly, in the View section, we keep all of our sprites which also have their own separate features. This allows us to quickly identify where things need to go as well as figuring out where we need to go to fix or add something.

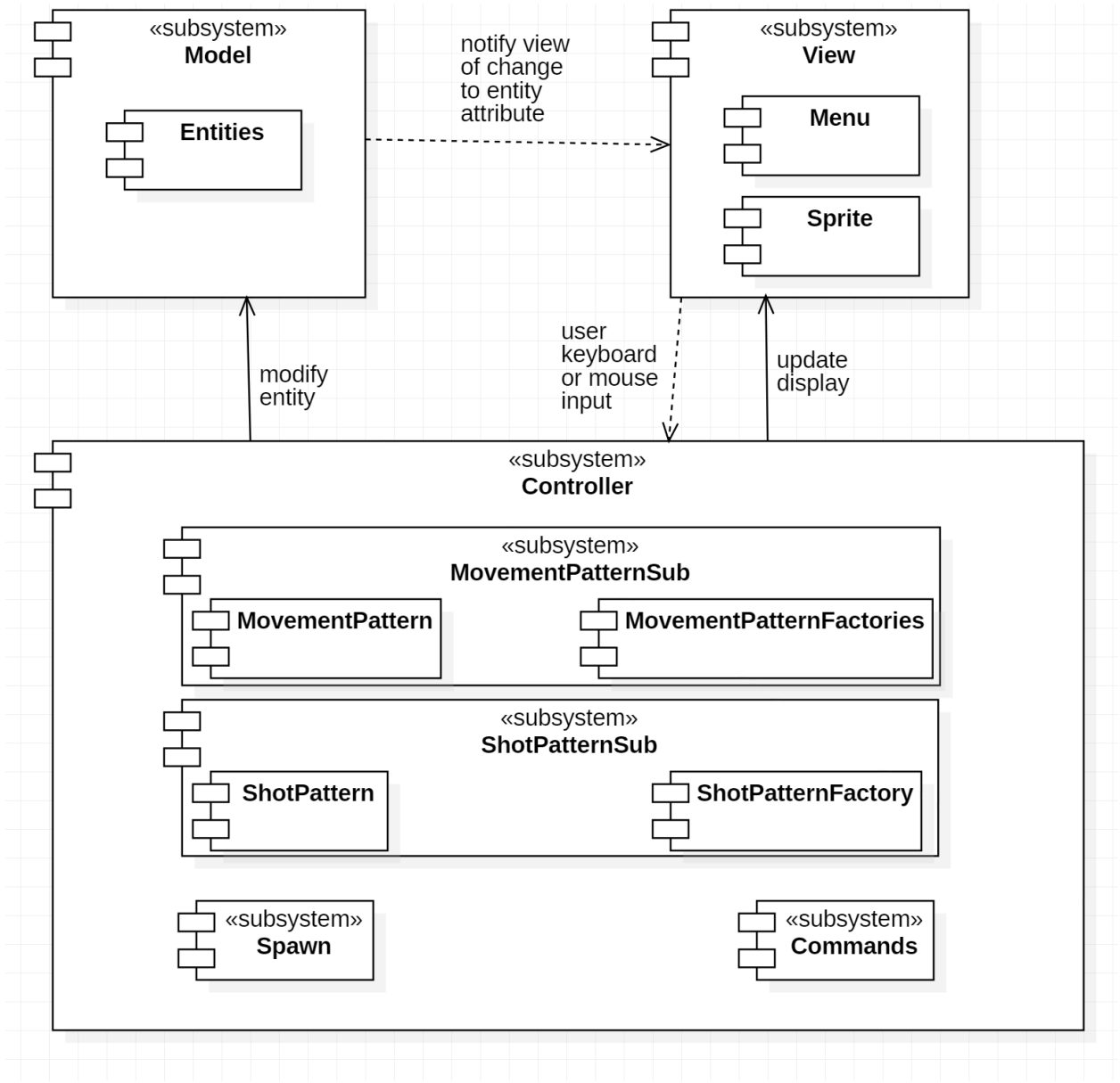

Figure 2.4: MVC Architectural Pattern

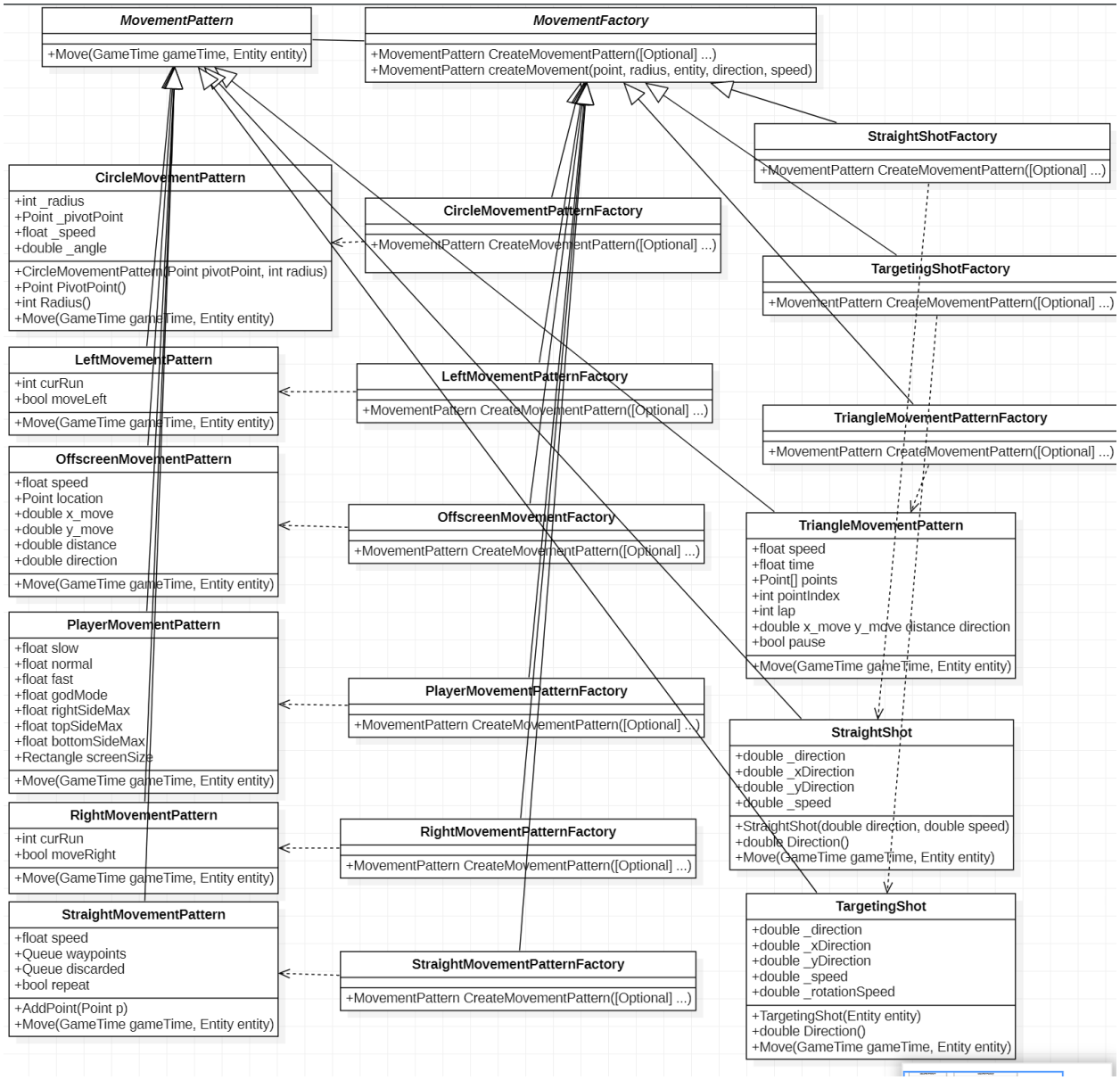

Figure 2.5: MovementPatternSub Subsystem

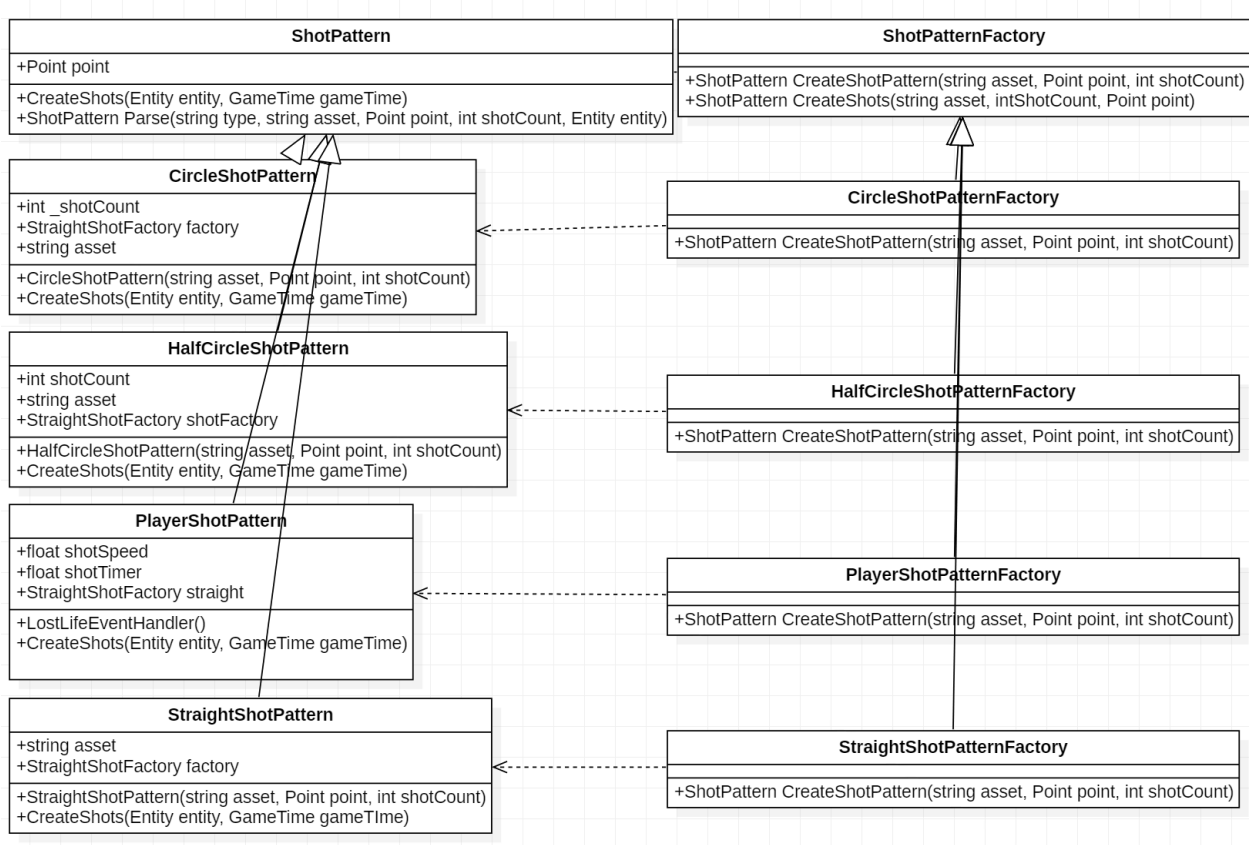

Figure 2.6: ShotPatternSub Subsystem

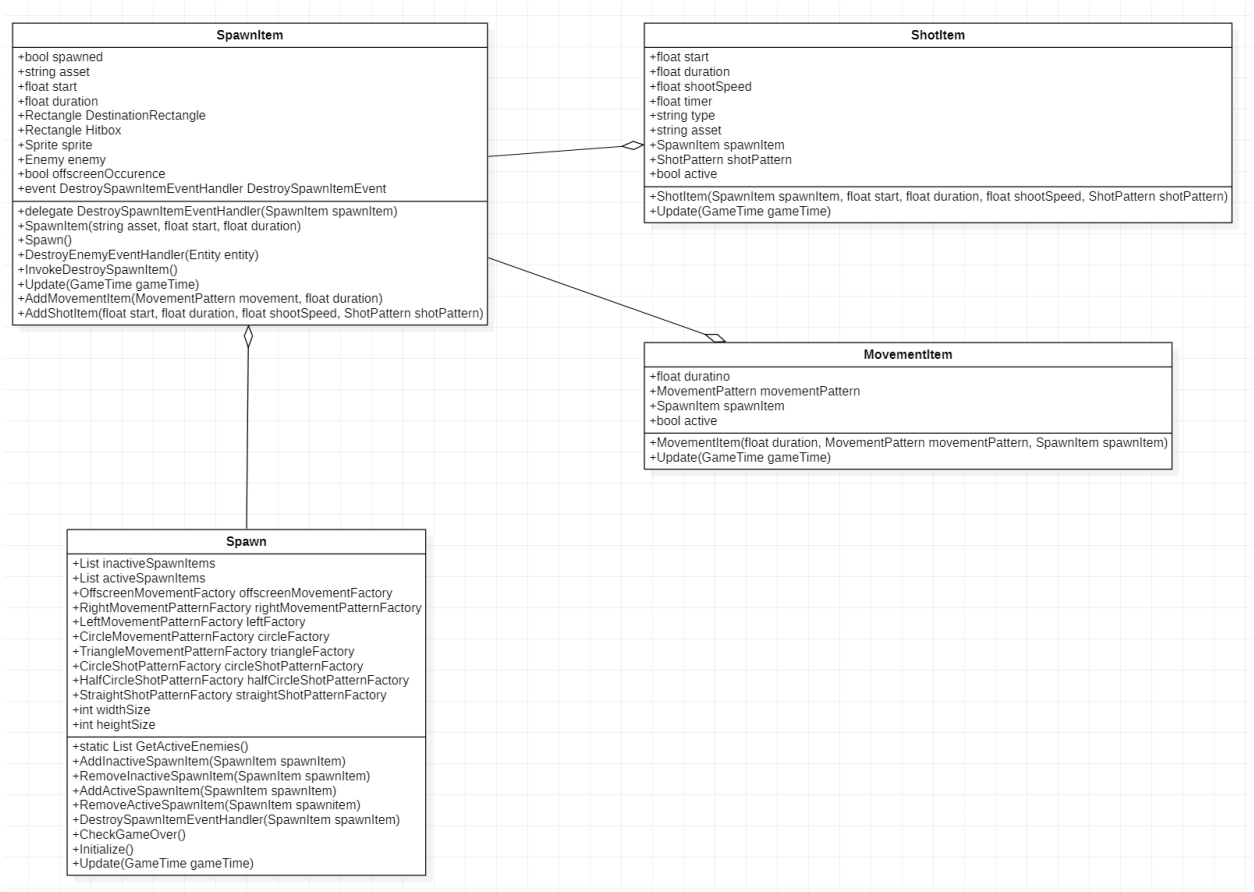

Figure 2.7: Spawn Subsystem

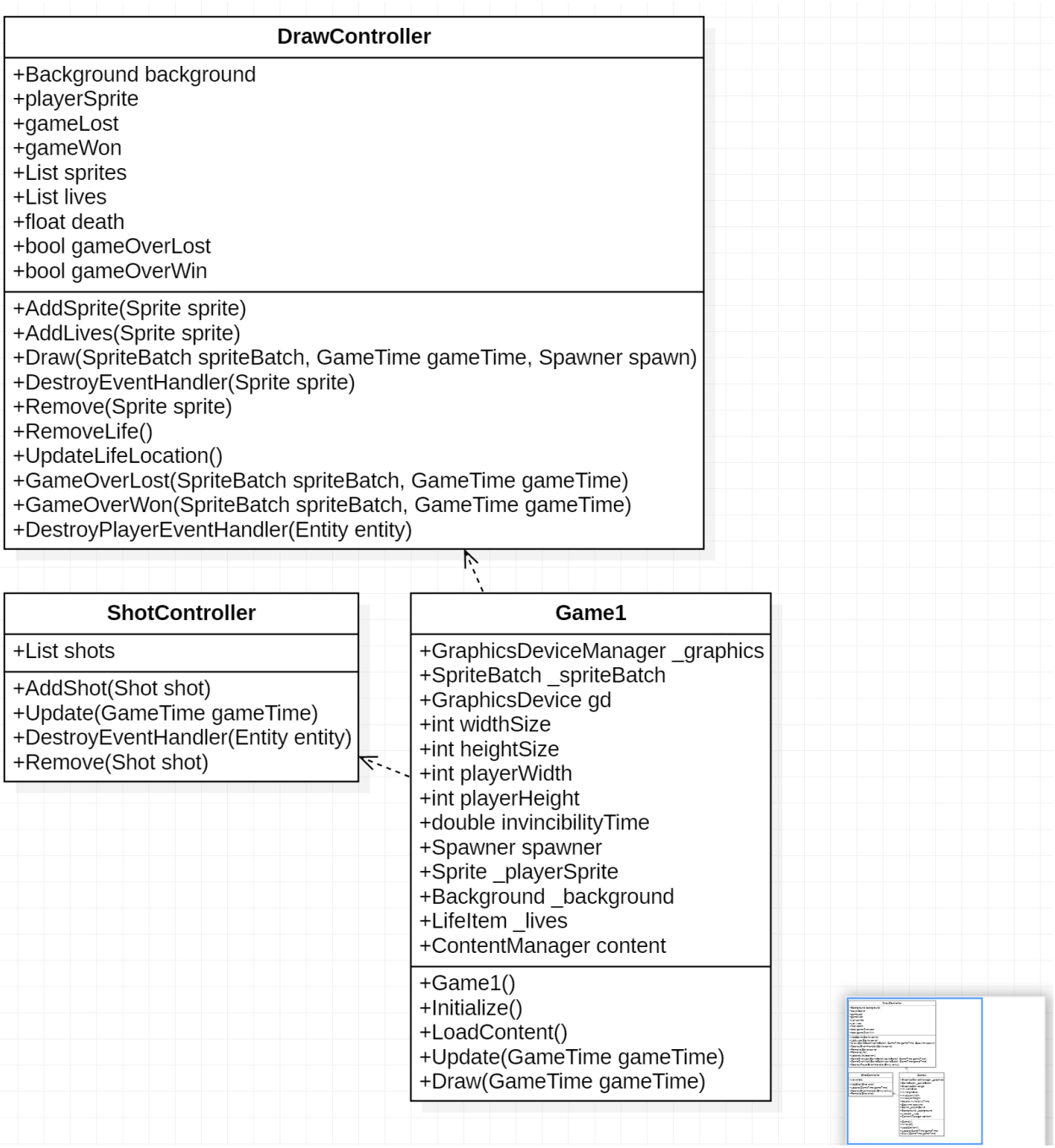

Figure 2.8: Controller Subsystem

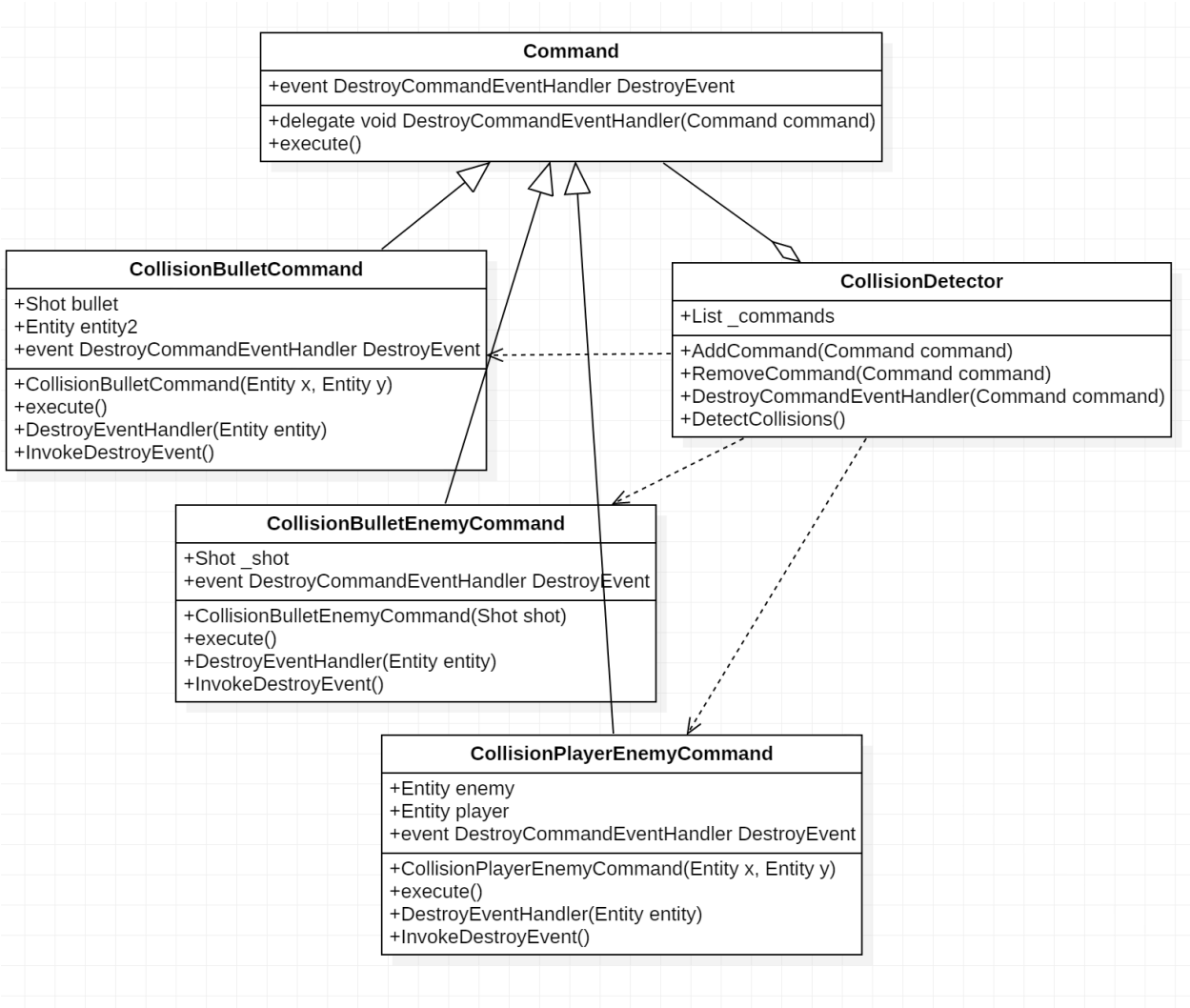

Figure 2.9: Commands Subsystem

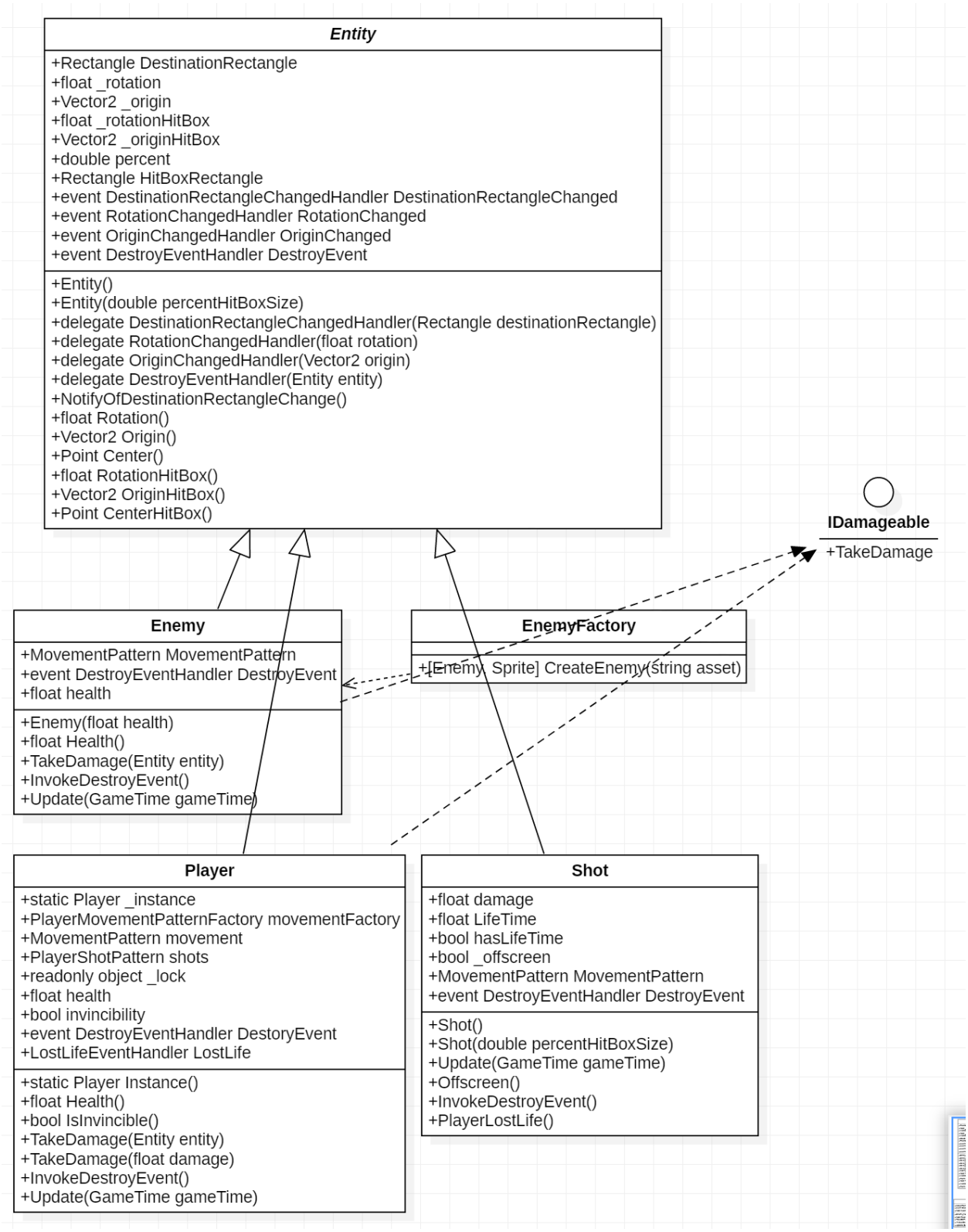

Figure 2.10: Model Subsystem

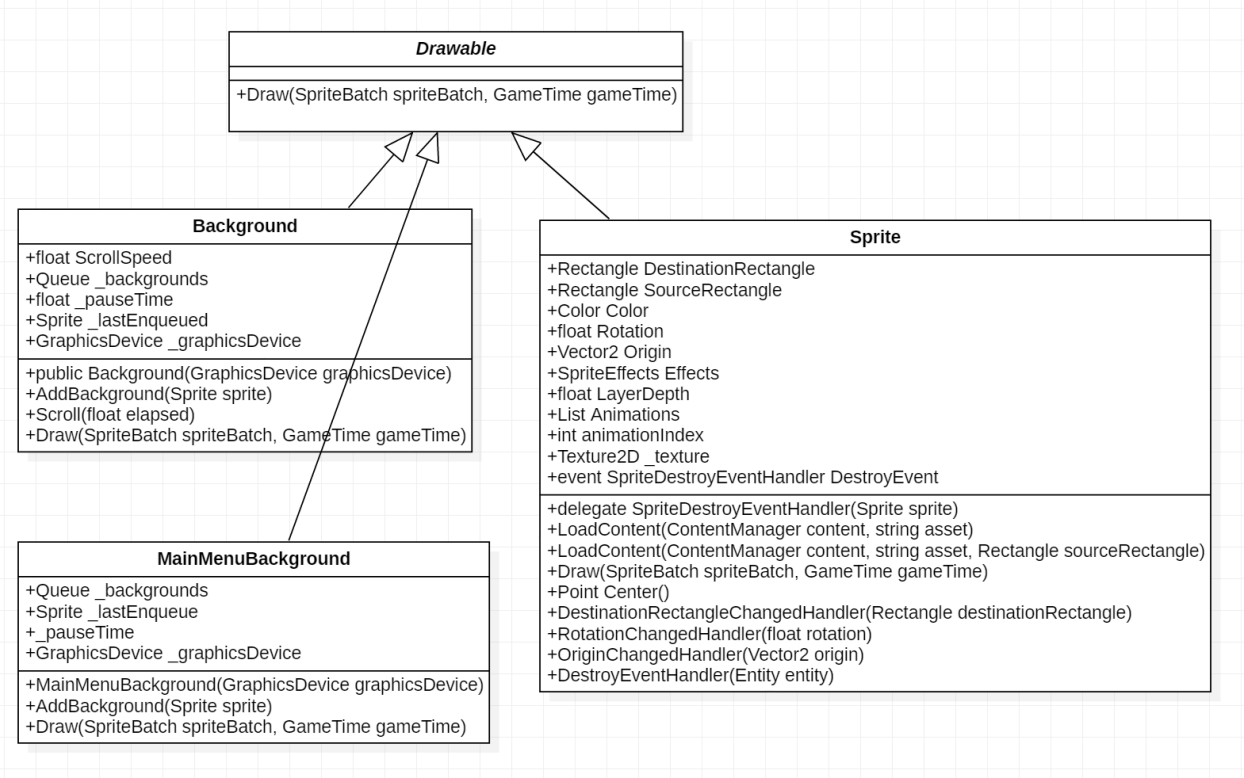

Figure 2.11: View Subsystem

2.2 Subsystem Decomposition

We grouped classes who stored only state information into the models subsystem, and classes who controlled or managed behavior in the controller subsystem, and classes that were purely for visual display in the view subsystem. Within each subsystem we created folders of related classes and then it made sense to group together some folders into their own subsystem, for sake of code organization we did not create a new folder representing these new subsystems. The two occasions of creating additional subsystems but not grouping them into a separate folder are: the MovementPatternSub and ShotPatternSub subsystems.

2.2.1 Controller

The controller subsystem is in charge of controlling the flow of the program, what that means is calling the update functions of objects that belong to subsystems within the controller or in the case of the view, outside of the controller subsystem. Everything starts in the Game1 which has an update function that updates all other subsystems that rely on a constant update.

2.2.2 MovementPatternSub

The MovementPatternSub subsystem defines MovementPatterns which can be applied to entities and provides interfaces through factory methods to create these MovementPatterns. MovementPatterns allow entities to move.

2.2.3 ShotPatternSub

The ShotPatternSub subsystem defines ShotPatterns which create shots in a predefined pattern such as a circle. The shots that are created are managed by the Controller subsystem. This subsystem provides an interface of factory methods for creating ShotPatterns, but remember the job of creating shots is that of the ShotPattern.

2.2.4 Spawn

The Spawn subsystem contains a series of classes that allow timing of enemies and the ability to let enemies have multiple shot and movement patterns throughout its lifetime. The Spawn subsystem makes the instantiation call for an enemy hence why it's called "Spawn". It has the power to bring to life and remove from life enemies.

2.2.5 Commands

The commands subsystem is our implementation of collision. When a shot is created, it must be encapsulated in a command and registered with the command subsystem for player-shot and enemy-shot collision. When an enemy is created, it must be encapsulated in a command and registered with the command subsystem for player-enemy collision.

2.2.6 Model

The model subsystem has a component called entities that contain classes that represent real objects or entities. The model contains a way for creating enemies that also returns the sprite since each entity should be associated with a sprite. The model subsystem provides an interface for creating new entities to other components.

2.2.7 View

The view subsystem contains visual elements such as a sprite, menu, and background. In the case of a sprite, it is associated with an entity model, but for example the background is purely a visual element with no helpful data tied to it.

2.2.8 Design Patterns

Factory Method: Factory Method is used in our project to create the different shot patterns as well as the movement patterns. This is useful for our project because instead of having a lot of "if" statements, we can now delegate them to factories which take the command and return back a new object based on inheritance.

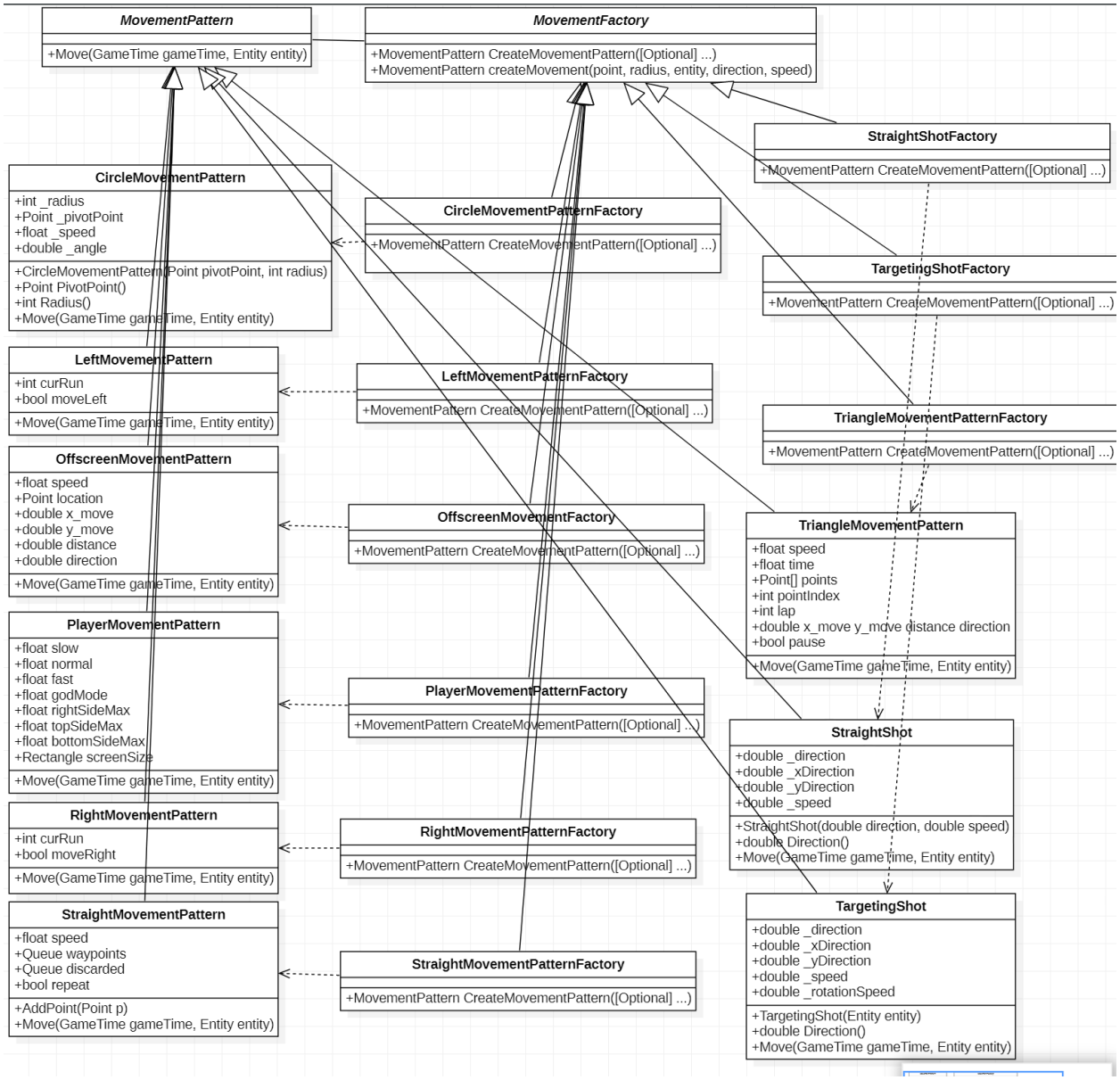

Figure 2.12: Factory Design Pattern

Command Pattern: The command pattern is used in our project to serve as the base template for our other objects to build on. The command pattern is useful because it allows us to create a base template called command which stores the base functions and then we use that to create our other commands since it inherits from that. We are currently using the command pattern for both collision detection and triggering the response to a collision.

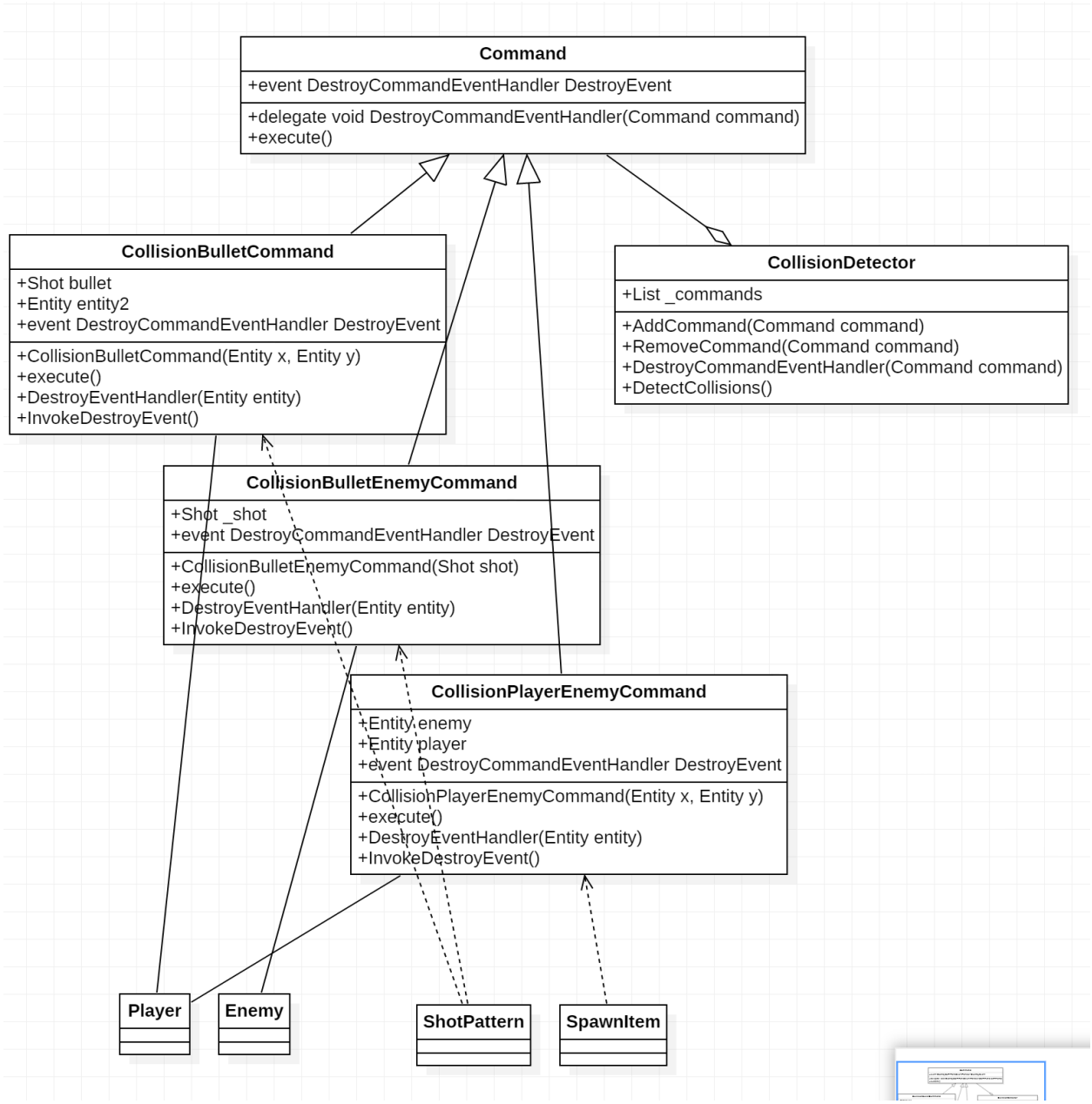

Figure 2.13: Command Design Pattern

Singleton: The singleton pattern is used in our project to serve the purpose of creating one instance of the player class. This is useful for us in the case where information about the player is needed and by having a singleton, we can explicitly declare it upon the first creation such that any time after that, the information will be static and can be grabbed from anywhere.

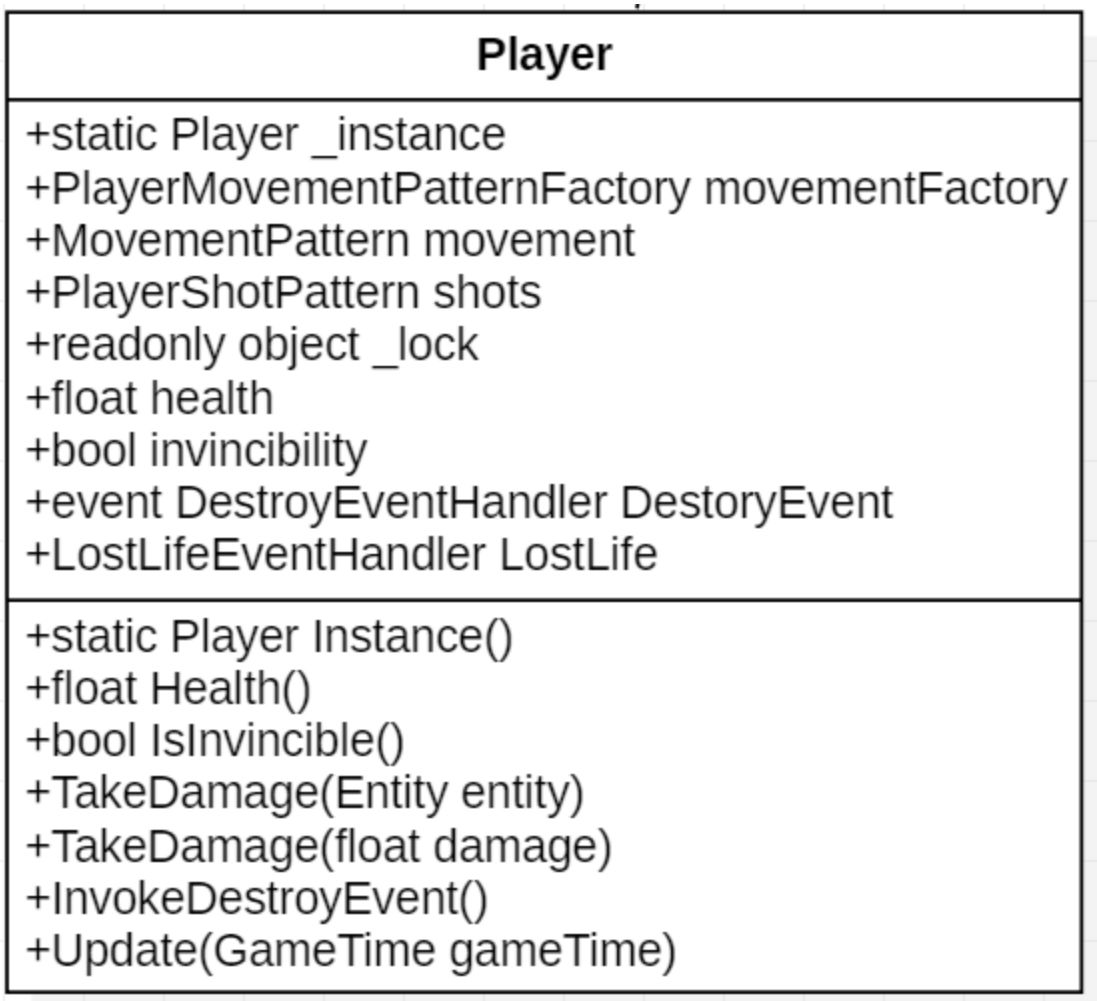

Figure 2.14: Singleton Design Pattern

Below is a description of all public implementations and their purpose within each component

Table 2.1: Player

Interface	Purpose
IsInvincible()	Returns a boolean indicating whether or not the player has invincibility frames
TakeDamage(entity)	Applies damage to the player after they have intersected with a “Shot” entity or they have intersected with an enemy character model
Update()	Performs a variety of operations, including handling user input to move the character on the screen, handling shots coming from the player, and also allows for gamepad input to

	move the player

Table 2.2: Enemy

Interface	Purpose
TakeDamage(entity)	Applies damage to the enemy after they have intersected with a “Shot” entity or they have intersected with the player character model
Update()	Moves the enemy given that a movement pattern is queued.

Table 2.3: Entity

Interface	Purpose
NotifyOfDestinationRectangleChange()	Recalculates HitBox of entity and invokes the DestinationRectangleChanged event

Table 2.4: Shot

Interface	Purpose
Offscreen()	Destroys shot if it is outside the bounds of the screen
Update()	Operates behavior of movement of bullet

Table 2.5: Button

Interface	Purpose
Clicked()	Begins the game
UpdateButton()	Displays button animation for duration of time

Table 2.6: IDamageable

Interface	Purpose
TakeDamage()	Overridden to apply damage to entity

Table 2.7: Spawner

Interface	Purpose
GetActiveEnemies()	Returns the list of enemies that are currently appearing on the screen.
AddInactiveSpawnItem()	Add a spawn item to the inactive spawn item list after they are no longer on the screen
RemoveInactiveSpawnItem()	Removes a spawn item from the inactive spawn item list after they enter the screen
AddActiveSpawnItem()	Adds a spawn item to the active spawn item list after it appears on screen
RemoveActiveSpawnItem()	Removes a spawn item from the active spawn list after they exit the screen
CheckGameOver()	Checks if there are any inactive or active spawn items left in the member variable lists. If there are none, the game is ended and the user has won
Initialize()	Initializes the model, movement pattern, and shot pattern for all entities that will be spawned during the course of the game's runtime
Update()	Ensures enemies enter/exit the screen at the correct timing intervals

Table 2.8: Spawn Item

Interface	Purpose
Spawn()	Assigns the entity their destination rectangle, draws the sprite of the entity, and adds them to the active spawn item list whilst removing them from the inactive spawn item list
AddMovementItem()	Adds a movement item to the entity that will be spawned
AddShotItem()	Adds a shot item to the entity that will be spawned
Update()	Spawns enemies, updates the actions of enemies (shooting, moving), and destroys

	them when they should exit the screen

Table 2.9: Shot Controller

Interface	Purpose
DestroyEventHandler()	Destroys shot
Remove()	Removes shot from shot list
AddShot()	Adds a shot to the shot list
Update()	Calls ShotItem ‘Update()’ function to handle the shot movement

Table 2.10: EnemyFactory

Interface	Purpose
CreateEnemy()	Handles the creation of an “Enemy” entity and returns the corresponding enemy and sprites that were created

Table 2.11: Sprite

Interface	Purpose
LoadContent()	Loads the sprite model that was passed from the ContentManager and bases the destination rectangle on the bounds of the model passed
Draw()	Draws the sprite to be displayed to the screen
DestinationRectangleChangedHandler()	Handled changing of destination rectangle
RotationChangedHandler()	Handles change in rotation value
OriginChangedHandler()	Handles change in origin value

Table 2.12: Background

Interface	Purpose
AddBackground()	Sets the background of the game based on the bounds of the screen given
Scroll()	Scrolls backgrounds if more than one are given
Draw()	Draws background sprite

Table 2.13: Mouse Button

Interface	Purpose
GetState()	Gets current state of the mouse
IsPressed()	Checks if mouse button is pressed down
HasNotBeenPressed()	Checks state of mouse button to see if it has not been pressed

Table 2.14: Enemy Controller

Interface	Purpose
AddEnemy()	Adds new enemy to member variable list of enemies
RemoveEnemy()	Removes enemy from list of enemies
Update()	Updates the current state of all the enemies in the member variable list of enemies

Table 2.15: Command

Interface	Purpose
Execute()	Abstract function that executes the command of the overriding class

Table 2.16: Collision Detector

Interface	Purpose
AddCommand()	Adds command to command list

RemoveCommand()	Removes the command from the command list
DetectCollisions()	Executes all commands that were stored in the command list

Table 2.17: CollisionBulletCommand

Interface	Purpose
Execute()	Checks whether the entity intersecting with the bullet is a player or an enemy and then applies the damage of the bullet as needed

Table 2.18: CollisionBulletEnemyCommand

Interface	Purpose
Execute()	Applies the damage of a bullet to an enemy

Table 2.19: CollisionPlayerEnemyCommand

Interface	Purpose
Execute()	Applies the damage the collision of an enemy and player to the player *only*

Table 2.20: Shot Pattern

Interface	Purpose
Parse()	Creates the correct shot pattern for an enemy based on the string passed
CreateShots()	Abstract function to be overridden to create shots

Table 2.21: Player Shot Pattern

Interface	Purpose
CreateShots()	Creates the correct shots for the player and queues them using the shot controller

Table 2.22: Half Circle Shot Pattern

Interface	Purpose
CreateShots()	Creates the correct shots for the entity which fire in front of the entity creating a half circle of bullets

Table 2.23: Circle Shot Pattern

Interface	Purpose
CreateShots()	Creates the correct shots for the entity, firing a circle of shots surrounding the entity

Table 2.24: Straight Shot Pattern

Interface	Purpose
CreateShots()	Creates the correct shots for the entity, firing shots in a straight pattern originating from the entity

Table 2.25: Shot Pattern Factory

Interface	Purpose
CreateShots()	Returns the result of the CreateShotsPattern function which was overridden by one of the specific shot pattern factory types
CreateShotPattern()	Abstract function which creates the correct shot pattern given the pattern desired

Table 2.26: Half Circle Shot Pattern Factory

Interface	Purpose
CreateShotPattern()	Creates the half circle shot pattern

Table 2.27: Circle Shot Pattern Factory

Interface	Purpose

CreateShotPattern()	Creates the circle shot pattern

Table 2.28: Straight Shot Pattern Factory

Interface	Purpose
CreateShotPattern()	Creates the straight shot pattern

Table 2.29: Movement Pattern

Interface	Purpose
Move()	Function to be overridden by inheriting classes to move an entity

Table 2.30: Left Movement Pattern

Interface	Purpose
Move()	Moves the entity to the left side of the screen, oscillating back to the middle of the screen

Table 2.31: Right Movement Pattern

Interface	Purpose
Move()	Moves the entity to the left side of the screen, oscillating back to the middle of the screen

3. Subsystem Services

Services are in bold. The ball is the provided interface while the line to the ball from a subsystem means that the subsystem depends on the service (interface between subsystems), or requires the interface.

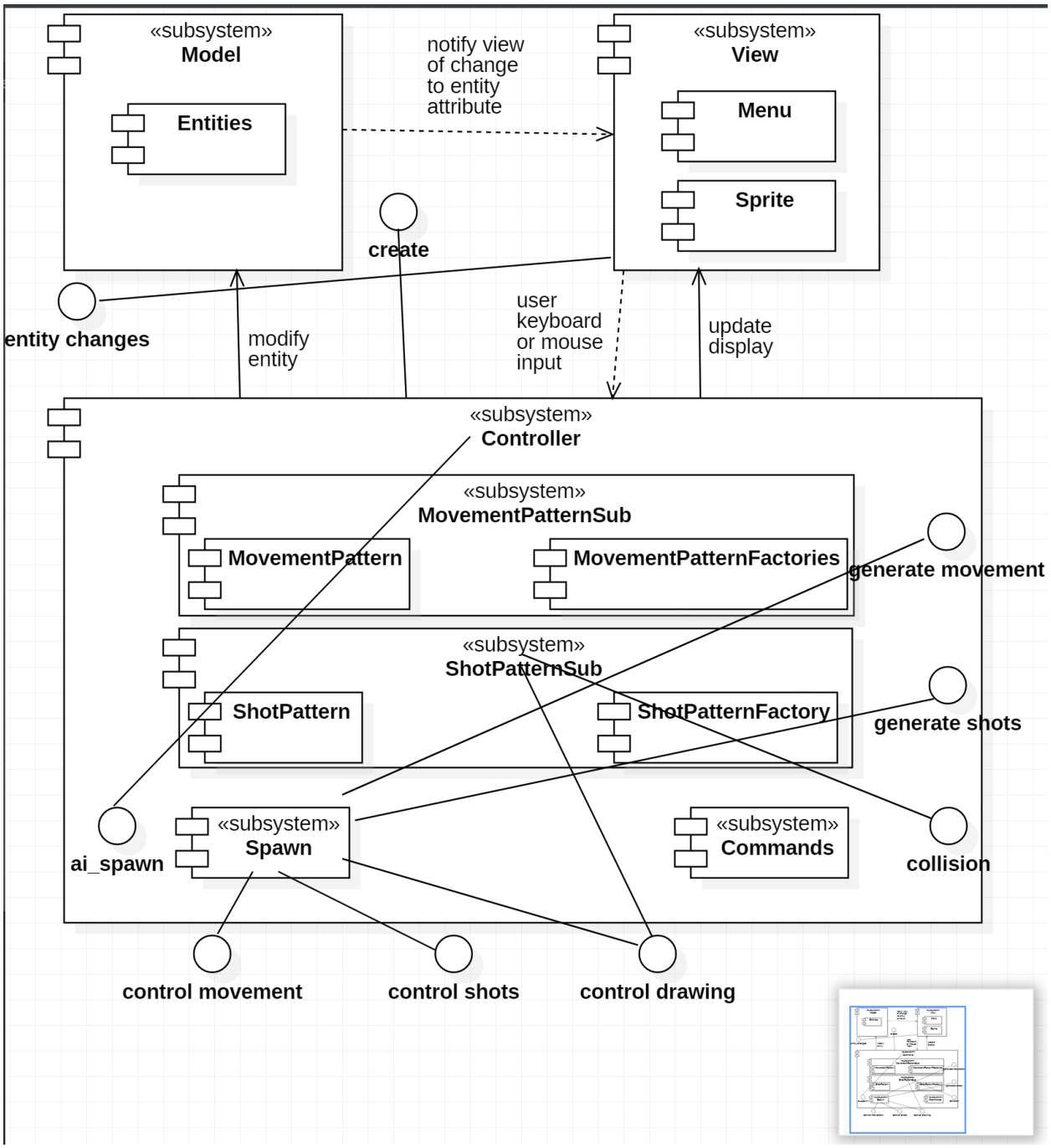

Figure 3.1: Subsystem Service Diagram

3.1 Controller Subsystem

The controller subsystem depends on the **create** service offered by the model subsystem by creating entities contained in the model using the Model's creation interface. The controller also provides services that manage drawing shots (**control drawing**) and controlling entity movement (**control shots** and **control movement**). The controller subsystem interacts with both the model and the view subsystem. It also contains the spawn, commands, movement pattern, and shot pattern subsystems. The controller has many subsystems within it that modify the properties of entities. An example of this is when the command subsystem changes the health of one of the entities contained within the model subsystem. The controller also updates the sprites based on the changes that are made in the many components within this subsystem, and therefore impacts the view subsystem as well.

3.2 MovementPattern Subsystem
The MovementPatternSub subsystem provides services for creating movement patterns which manage the movement of entities (**generate movement**). Additionally, the movement pattern subsystem is contained within the controller subsystem. It depends on the model subsystem for an entity to move. Entities are given movement patterns which change their properties by allowing them to move across the screen in various different ways. This in turn impacts the view subsystem, when the entities are changed.

3.3 ShotPattern Subsystem
The ShotPatternSub subsystem provides services for creating shot patterns which manage spawning shots (**generate shots**). The ShotPatternSub subsystem depends on the interfaces provided by the Model to create shots.The shot pattern subsystem is also contained within the controller subsystem. This subsystem also directly impacts the model, by changing the shot properties of the entities contained within that subsystem.

3.4 Spawn Subsystem
This subsystem spawns entities, and thus relies on the model's services. Entities are created to be displayed on the screen, and thus it also interacts with the view subsystem. This subsystem is contained within the controller subsystem. The spawn subsystem provides services for spawning enemies which are required to start a game (**ai_spawn**).

3.5 Commands Subsystem
The commands subsystem relies on the model subsystem for an entity to track for collision. The commands subsystem provides a service to models that notifies them of collision and can be expanded to support other services (**collision**). This subsystem is contained within the controller subsystem.

3.6 Model Subsystem
The model subsystem provides services that notify the view of any changes in the model so that the view can react accordingly (**entity changes**). The model subsystem is modified by the controller, which will make changes to the state of an entity, and the entity is expected to respond accordingly. For example, if an enemy is destroyed in the EnemyController class, that corresponding enemy in the model subsystem would be destroyed. The model subsystem also

provides an interface for creating models, right now for enemies only since the player is a singleton and shots are created in the ShotPattern (**create**).

3.7 View Subsystem

The view subsystem interfaces with the model subsystem by receiving information from the model on change of properties of an entity within the view subsystem. For example, the view is told when an enemy is moving so that the sprite can also move. It also receives information from the controller subsystem on when to update the display. An example of this is the DrawController, which adds sprites to be drawn.