Software Design Document

for

Bo & Long Go to Panda Express

Team: TBD

Project: Bo & Long Go to Panda Express

Team Members:
Antonio Barber
Emily Cawlfield
Aaron Holbrook
Alexander McKee
Aric Monary
Kristian Suzara

Table of Contents

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
0.1	3/25/2021	Initial draft	Initial draft gathers all ideas to distribute the creation of required figures.
0.2	3/28/2021	First review draft	Draft for review by team before submission.
0.3	3/28/2021	Post-review changes part 1	The review outcome determined several impactful changes were necessary.
0.4	3/31/2021	Post-review changes part 2	Continuation of necessary changes and updates based on code refactorings.
1.0	4/4/2021	Finished document	After final edits, the document was deemed acceptable for submission.

List of Figures

List of Tables

1. Introduction

Bo & Long Go to Panda Express is a bullet hell game based on the popular Touhou franchise. The premise of the game is to dodge projectiles launched by enemies and to make it to the end of all the waves with at least one life remaining.

1.1 Architectural Design Goals

1.1.1 Modifiability

The game's wave content needed to be easily changed without directly modifying the source code. This was to allow for quick and easy creation/modification of different waves in the game, ranging from enemy type, projectiles they fired, how they moved, where they would spawn, and when they would spawn in terms of waves. At first, the system had hard coded strings put into dictionaries that followed a specific structure, this would be the basis of the JSON-originated dictionaries that are loaded into the game system. To allow use of the dictionary structure outside of the source code, a JSON parsing system had to be implemented to allow storage of Wave and game configuration info in files that could be loaded into the system that it would interpret.

The structure of the dictionaries has been changed multiple times through the course of development to simplify them further. For example, in an early iteration, each enemy had to be defined separately, but then a later change allowed for spawning singular types of enemies in a group. Those changes, on top of the abstract structure the majority of the classes follow, shortened the overall process of creating the JSON files and the length of text needed in such files. The abstract class structure for the characters and their attacks starts from Sprites, then extends down to more specialized abstract classes such as the Entity and Projectile classes. This allows all classes to share a small number of characteristics as they are all Sprites, but then it becomes more focused going down the inheritance tree. The Composite structural design pattern was used between entities and projectiles, which contribute to the ease of use of the JSONs as well. Together, these allow for each Entity to follow a set structure as follows: entity type, texture, color, movement, and their projectile. Overall, without the set structure and abstract extendability, the JSON files would be much more time consuming to create or modify.

1.1.2 Performance

To maintain reasonable performance goals for this project, i.e. ensuring 60 frames could be drawn per second, the team focused on managing the amount of updates needed per frame as well as keeping their computational intensity relatively low. The design choices required in collision detection in particular necessitated updating for this reason, as the initial attempt at handling such a process from the game management side instead of in individual entity classes lessened performance to an unacceptable level much lower than 60 frames per second. To fix this, instead of doing a narrow-phase, high-accuracy, per-pixel collision detection for all entities every frame, it was decided that rectangular hitbox-based collision was sufficient for the team's purposes. Other design choices in line with achieving reasonable performance goals were those that affected how many different entity classes there were and their hierarchy, since those would be the ones needing updates per frame. This included making use of abstraction and composition so that effort needn't be put into managing separate lists of entities and sprites, or determining which type was being handled most of the time, with operations and attributes made common through an abstract class.

2. Software Architecture

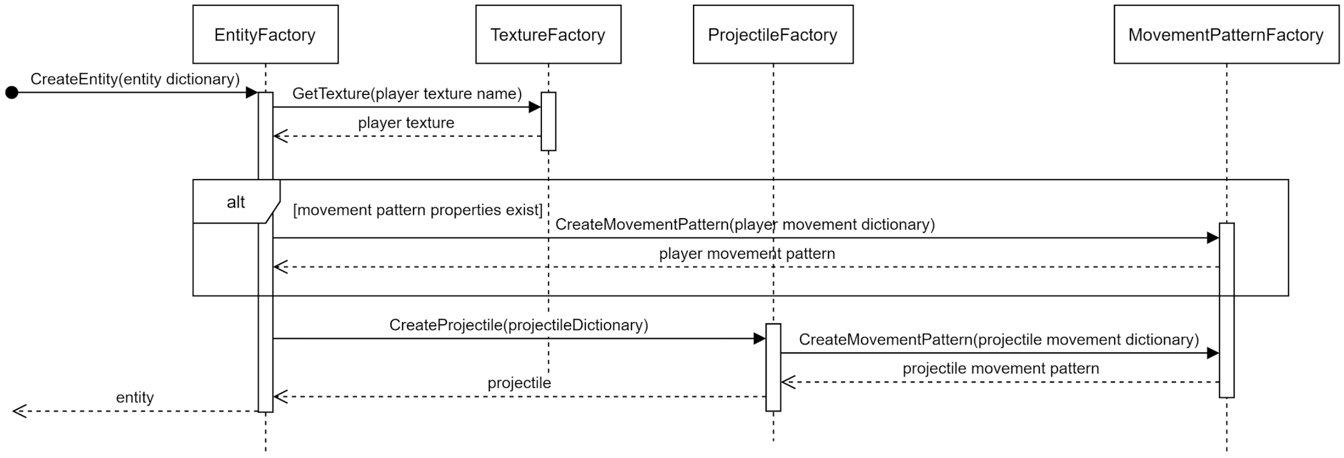
EntityFactory
TextureFactory
ProjectileFactory
MovementPatternFactory
CreateEntity(entity dictionary)
GetTexture(player texture name)
player texture
alt
[movement pattern properties exist]
CreateMovementPattern(player movement dictionary)
player movement pattern
CreateProjectile(projectileDictionary)
CreateMovementPattern(projectile movement dictionary)
projectile movement pattern
projectile
entity

Figure 2-1: Entity Creation Sequence Diagram

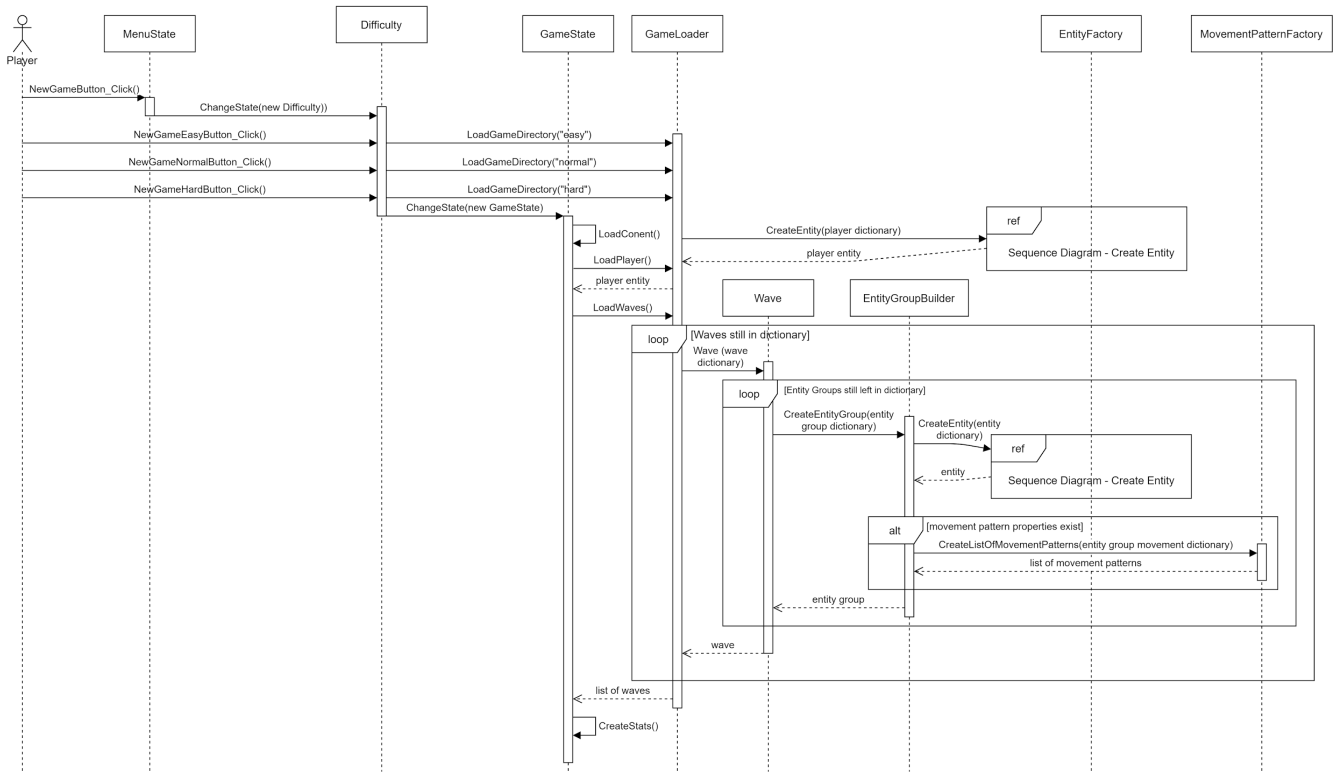
Player
MenuState
Difficulty
GameState
GameLoader
EntityFactory
MovementPatternFactory
NewGameButton_Click()
ChangeState(new Difficulty))
NewGameEasyButton_Click()
LoadGameDirectory("easy")
NewGameNormalButton_Click()
LoadGameDirectory("normal")
NewGameHardButton_Click()
LoadGameDirectory("hard")
ChangeState(new GameState)
LoadConent()
CreateEntity(player dictionary)
ref
Sequence Diagram - Create Entity
player entity
LoadPlayer()
player entity
Wave
EntityGroupBuilder
LoadWaves()
loop
[Waves still in dictionary]
Wave (wave dictionary)
loop
[Entity Groups still left in dictionary]
CreateEntityGroup(entity group dictionary)
CreateEntity(entity dictionary)
ref
Sequence Diagram - Create Entity
entity
alt
[movement pattern properties exist]
CreateListOfMovementPatterns(entity group movement dictionary)
list of movement patterns
entity group
wave
list of waves
CreateStats()

Figure 2-2: Game Setup Sequence Diagram (refers to Figure 2-1)

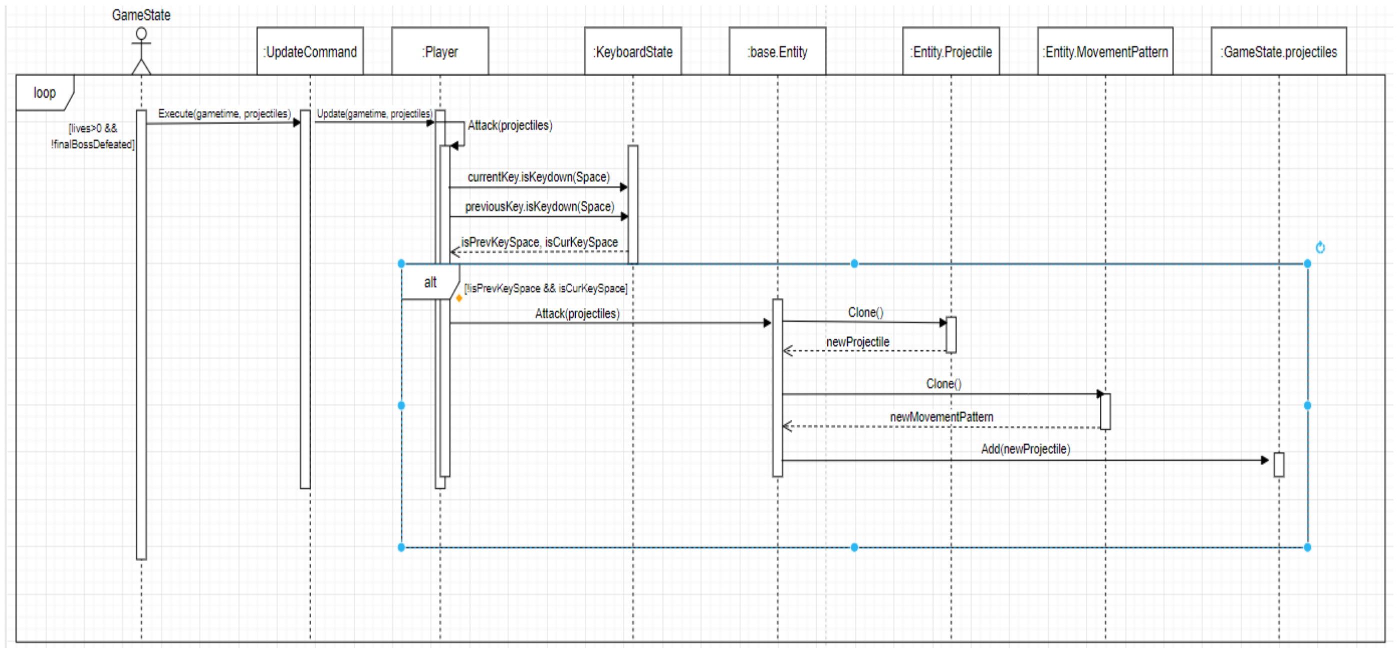

Figure 2-3: Player Attack Sequence Diagram

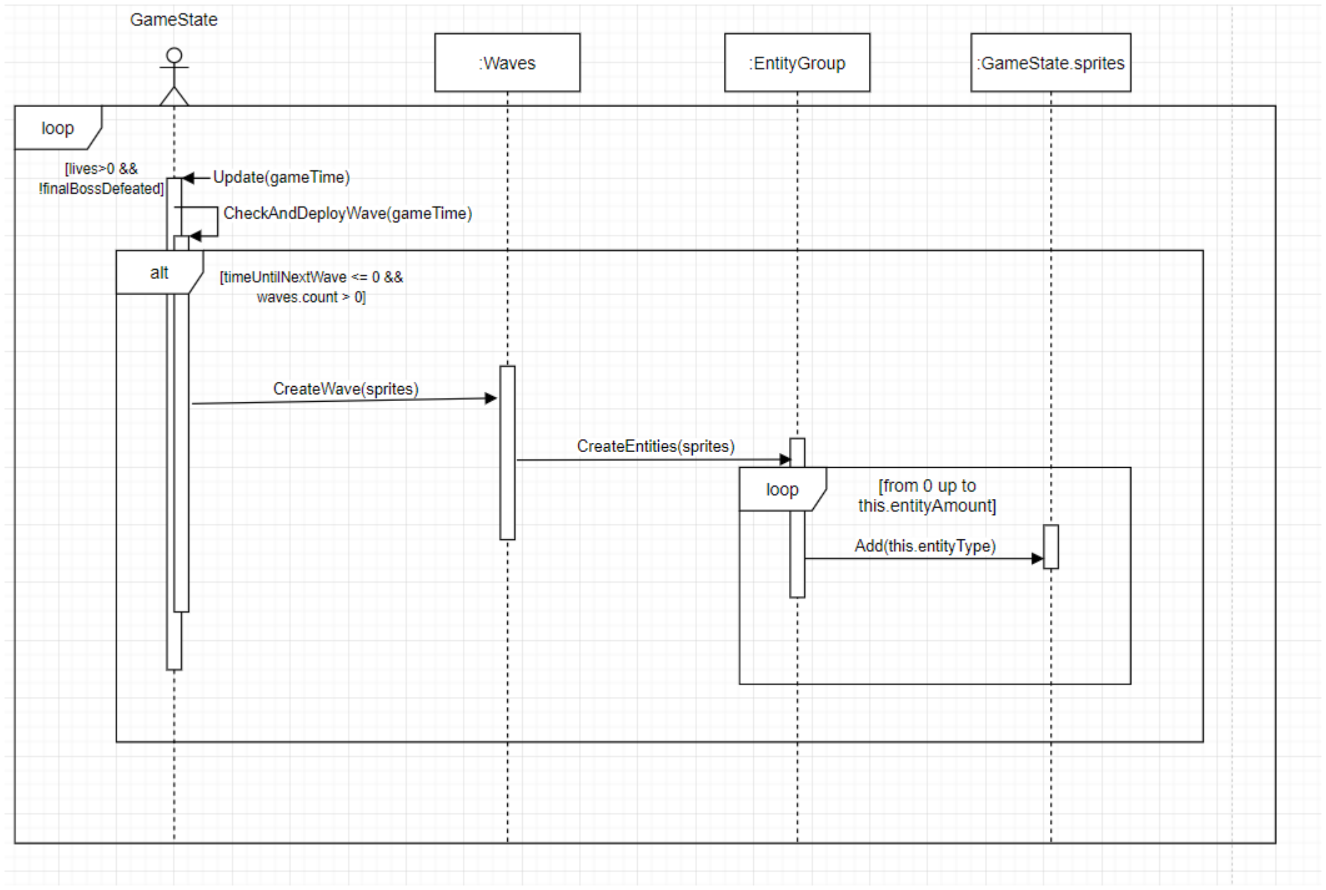

Figure 2-4: Wave Spawn Sequence Diagram

2.1 Overview

The architectural pattern the team has chosen is Multi-Layered. This pattern was chosen as this game can be best broken down into 3 tiers: Presentation, Logic, and Data. The Presentation layer is for display purposes and is what the player interacts with. Within this layer is the GUI subsystem, which handles the drawing of everything on the screen and displaying the different states. The Logic layer is for managing what needs to be displayed in the presentation layer, as well the interactions between those items displayed. The Logic layer contains the State Management subsystem, which is what directly informs the GUI which state needs to be displayed, as well as what Sprite-related updates to display. This layer also contains the Sprite Management subsystem, which supports State Management by providing collision detection and Sprite interaction-related updates. The Data layer handles the JSON information and interprets that information for use within the logic layer. This layer contains the Game Setup subsystem, which reads in JSON files containing game configuration details and provides the State Management system with Wave and Sprite creation information by way of a game dictionary.

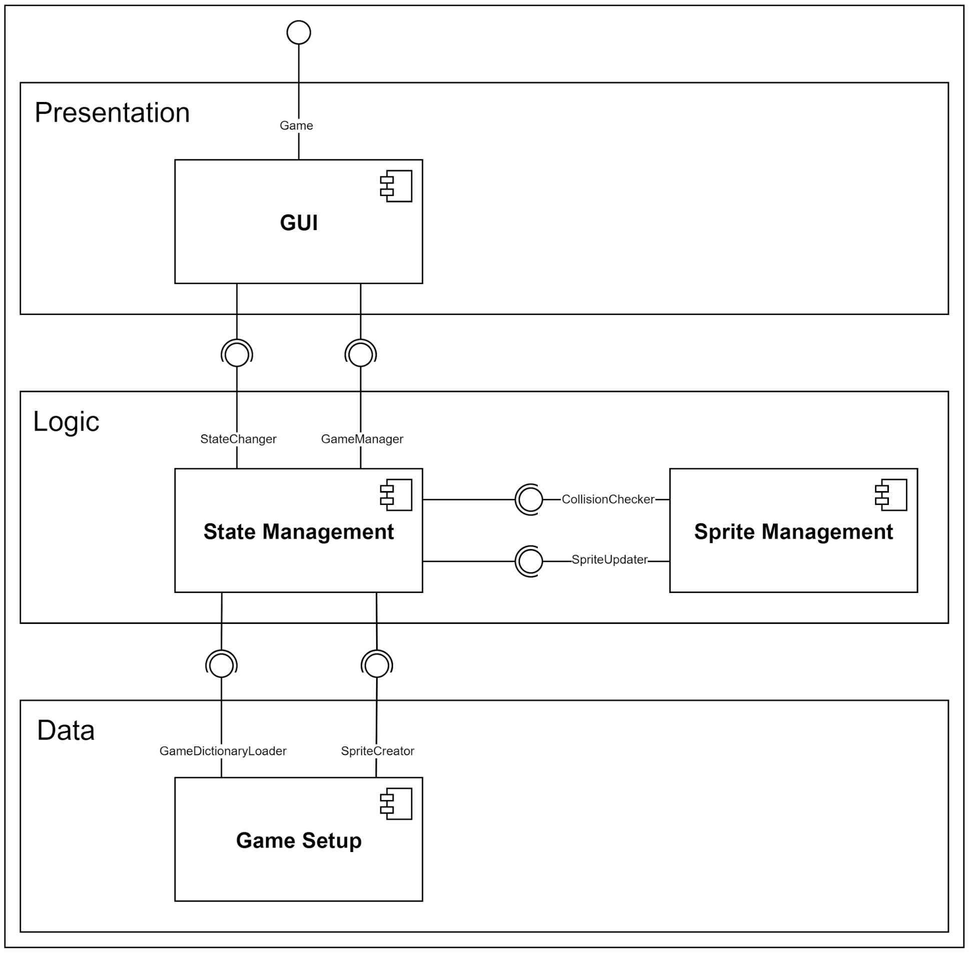

Figure 2-5: Multi-layered Architecture Component Diagram

2.2 Subsystem Decomposition

2.2.1 State Management

The Menu subsystem supports the user interface that allows users to configure or start a game. The different menu states in the game are: Main Menu, Options, Configure Keys, Select Difficulty, Game, Game Over Win, and Game Over Lose. These states allow the users to switch between the different menu states, in addition to transitioning to and from the Game state. Inside of the GUI class, the StateManager's current state will be called to use its Draw and Update methods for rendering. These Update and Draw functions are defined in the State abstract class, and overridden by each State subclass. The Main Menu state is the first state loaded in when the game is started. From the Main Menu state, the user can transition to the Select Difficulty menu state, the Options menu state, or exit the game. From the Options menu state, the user can transition to the Configure Controls menu state or return to the Main Menu state. The Configure Controls state is where the user can rebind the keys to control the player in the game or return to the Options Menu state. The Select Difficulty menu state is where the user can select a difficulty. When a difficulty is selected, the GameLoader.LoadGameDictionary is passed a JSON file as an argument to parse and load. The specific JSON file is based upon which difficulty is selected. After the JSON file is passed to the GameLoader.LoadGameDictionary, the Select Difficulty state transitions to the Game state. The Game state is where the game is loaded and the user can play it. If the user has zero lives left, then the state switches to the Game Over Lose state. In the Game Over Lose state, the words "GAME OVER" are displayed, and the player is prompted with the option to play again (which loads the select difficulty state), exit to the Main Menu, or quit. If the user defeats the final boss or the final boss despawns in the game, then the state switches to the Game Over Win state. In the Game Over Win state, the words "YOU WIN!" are displayed, and the player is prompted with the option to play again (which loads the select difficulty state), exit to the Main Menu, or quit the game.

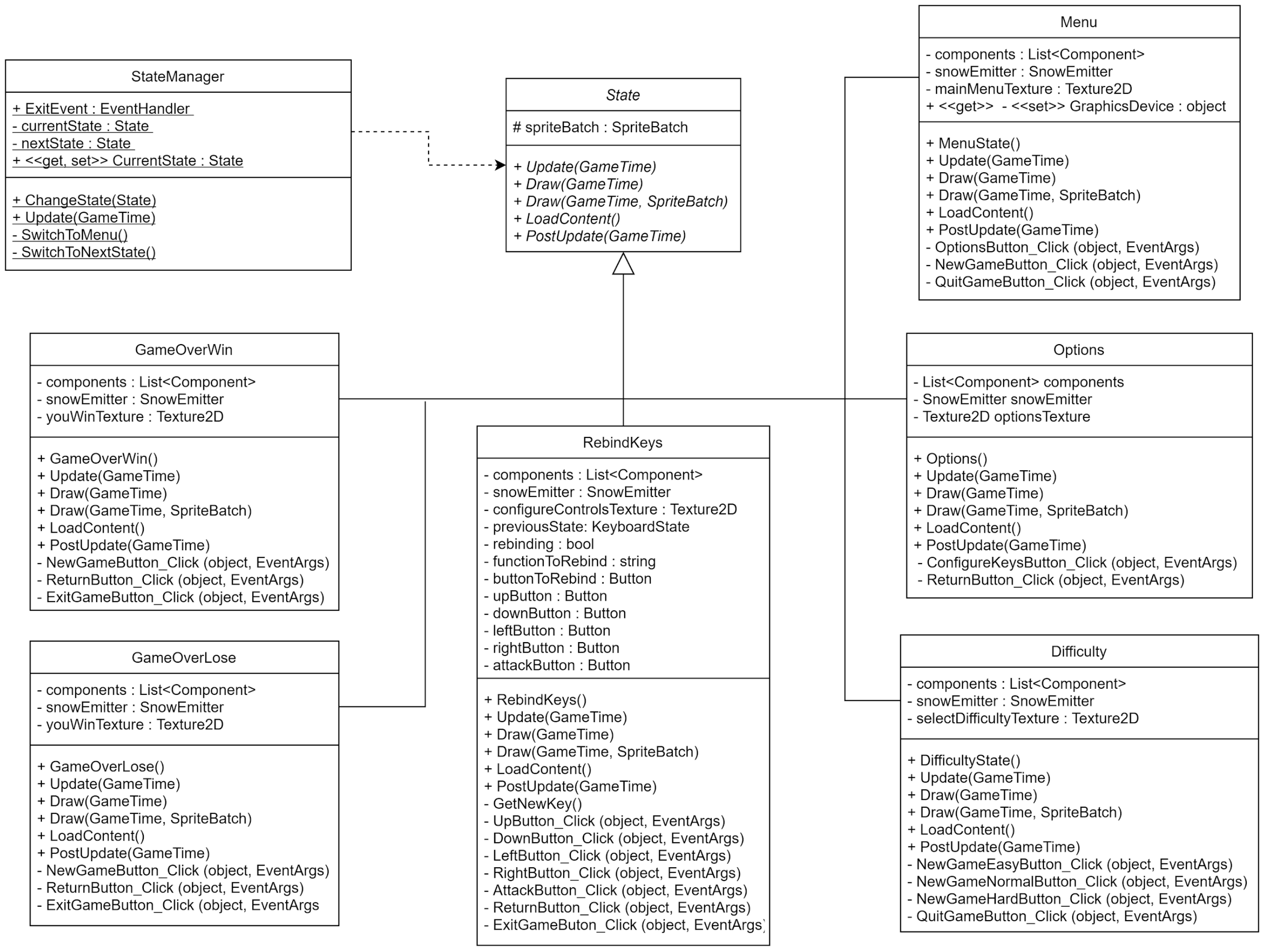
StateManager
+ ExitEvent : EventHandler
- currentState : State
- nextState : State
+ <<get, set>> CurrentState : State
+ ChangeState(State)
+ Update(GameTime)
- SwitchToMenu()
- SwitchToNextState()
State
# spriteBatch : SpriteBatch
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
Menu
- components : List<Component>
- snowEmitter : SnowEmitter
- mainMenuTexture : Texture2D
+ <<get>> - <<set>> GraphicsDevice : object
+ MenuState()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- OptionsButton_Click (object, EventArgs)
- NewGameButton_Click (object, EventArgs)
- QuitGameButton_Click (object, EventArgs)
GameOverWin
- components : List<Component>
- snowEmitter : SnowEmitter
- youWinTexture : Texture2D
+ GameOverWin()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- NewGameButton_Click (object, EventArgs)
- ReturnButton_Click (object, EventArgs)
- ExitGameButton_Click (object, EventArgs)
RebindKeys
- components : List<Component>
- snowEmitter : SnowEmitter
- configureControlsTexture : Texture2D
- previousState: KeyboardState
- rebinding : bool
- functionToRebind : string
- buttonToRebind : Button
- upButton : Button
- downButton : Button
- leftButton : Button
- rightButton : Button
- attackButton : Button
+ RebindKeys()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- GetNewKey()
- UpButton_Click (object, EventArgs)
- DownButton_Click (object, EventArgs)
- LeftButton_Click (object, EventArgs)
- RightButton_Click (object, EventArgs)
- AttackButton_Click (object, EventArgs)
- ReturnButton_Click (object, EventArgs)
- ExitGameButon_Click (object, EventArgs)
Options
- List<Component> components
- SnowEmitter snowEmitter
- Texture2D optionsTexture
+ Options()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- ConfigureKeysButton_Click (object, EventArgs)
- ReturnButton_Click (object, EventArgs)
GameOverLose
- components : List<Component>
- snowEmitter : SnowEmitter
- youWinTexture : Texture2D
+ GameOverLose()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- NewGameButton_Click (object, EventArgs)
- ReturnButton_Click (object, EventArgs)
- ExitGameButton_Click (object, EventArgs
Difficulty
- components : List<Component>
- snowEmitter : SnowEmitter
- selectDifficultyTexture : Texture2D
+ DifficultyState()
+ Update(GameTime)
+ Draw(GameTime)
+ Draw(GameTime, SpriteBatch)
+ LoadContent()
+ PostUpdate(GameTime)
- NewGameEasyButton_Click (object, EventArgs)
- NewGameNormalButton_Click (object, EventArgs)
- NewGameHardButton_Click (object, EventArgs)
- QuitGameButton_Click (object, EventArgs)

Figure 2-6: State Management Subsystem Class Diagram

2.2.2 Game Setup

The game setup subsystem primary interface is the GameLoader class which is used by the GameState and the Difficulty State. The responsibilities of the Game Setup system is to interpret a JSON that corresponds to the selected difficulty and then create waves and the player based on the interpreted JSON. The interpretation involves converting the JSON to a dictionary that can be programmatically evaluated by the factories to create waves and the player. The JSON interpretation occurs when the difficulty is selected in the menu. The wave creation occurs in the load content sequence of the GUI class.

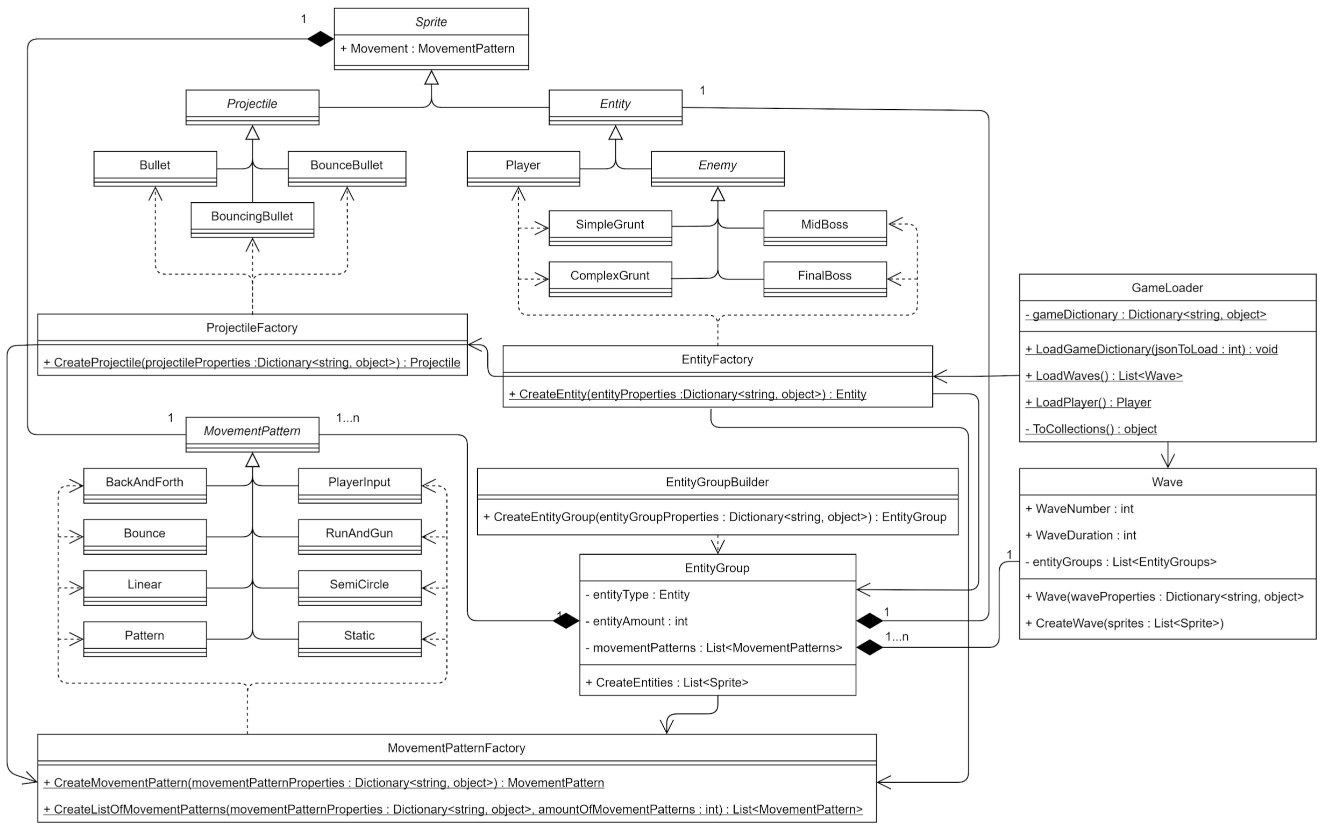
Sprite
+ Movement : MovementPattern
Projectile
Entity
Bullet
BounceBullet
BouncingBullet
Player
Enemy
SimpleGrunt
MidBoss
ComplexGrunt
FinalBoss
ProjectileFactory
+ CreateProjectile(projectileProperties :Dictionary<string, object>) : Projectile
EntityFactory
+ CreateEntity(entityProperties :Dictionary<string, object>) : Entity
GameLoader
- gameDictionary : Dictionary<string, object>
+ LoadGameDictionary(jsonToLoad : int) : void
+ LoadWaves() : List<Wave>
+ LoadPlayer() : Player
- ToCollections() : object
MovementPattern
BackAndForth
PlayerInput
Bounce
RunAndGun
Linear
SemiCircle
Pattern
Static
EntityGroupBuilder
+ CreateEntityGroup(entityGroupProperties : Dictionary<string, object>) : EntityGroup
EntityGroup
- entityType : Entity
- entityAmount : int
- movementPatterns : List<MovementPatterns>
+ CreateEntities : List<Sprite>
Wave
+ WaveNumber : int
+ WaveDuration : int
- entityGroups : List<EntityGroups>
+ Wave(waveProperties : Dictionary<string, object>
+ CreateWave(sprites : List<Sprite>)
MovementPatternFactory
+ CreateMovementPattern(movementPatternProperties : Dictionary<string, object>) : MovementPattern
+ CreateListOfMovementPatterns(movementPatternProperties : Dictionary<string, object>, amountOfMovementPatterns : int) : List<MovementPattern>
1
1
1
1...n
1
1
1
1...n

Figure 2-7: Game Setup Subsystem Class Diagram

2.2.3 Sprite Management

The Sprite Management subsystem contains 3 classes: ICommand, CollisionCheckCommand, UpdateCommand, which allow the GameState class to interact with Sprite logic less directly. UpdateCommand encapsulates the Sprite's Update method, which can involve smaller actions such as moving, attacking, respawning, or checking lifespan timing, in such a way that it can be done within a queue of other commands as necessary. The CollisionCheckCommand keeps the logic for checking whether certain sprite groups have collided with other sprites (such as the player, projectiles, or enemies) out of the GameState class. Despite the name of this subsystem, it is to be noted that the responsibility for Sprite creation belongs to the Game Setup system, and Sprite death is currently handled as part of State Management.

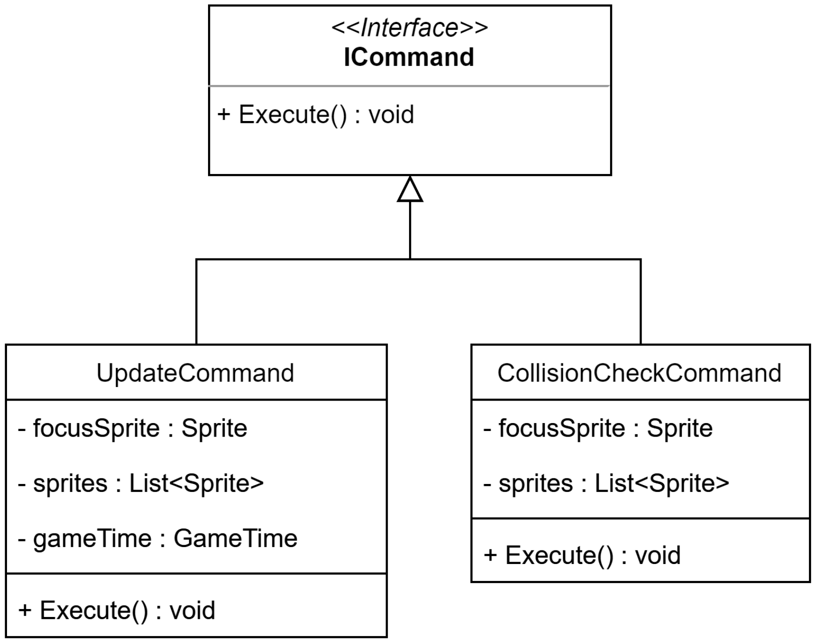

Figure 2-8: Sprite Management Subsystem Class Diagram

2.3 Design Patterns Used

2.3.1 Factory Method Pattern for Entity, Projectile, and Movement Pattern Creation

Factory Method Pattern Mapping:

- Products: Entity, Projectile, and MovementPattern
- Concrete Products:
  - Entity: Player, SimpleGrunt, ComplexGrunt, MidBoss, and FinalBoss
  - Projectile: Bullet, BounceBullet, BouncingBullet
  - MovementPattern: BackAndForth, Bounce, LinearPattern, PlayerInput, RunAndGun, SemiCircle, and Static
- Concrete Creators: EntityFactory, ProjectileFactory, and MovementPatternFactory
- FactoryMethod: CreateEntity, CreateProjectile, CreateMovementPattern

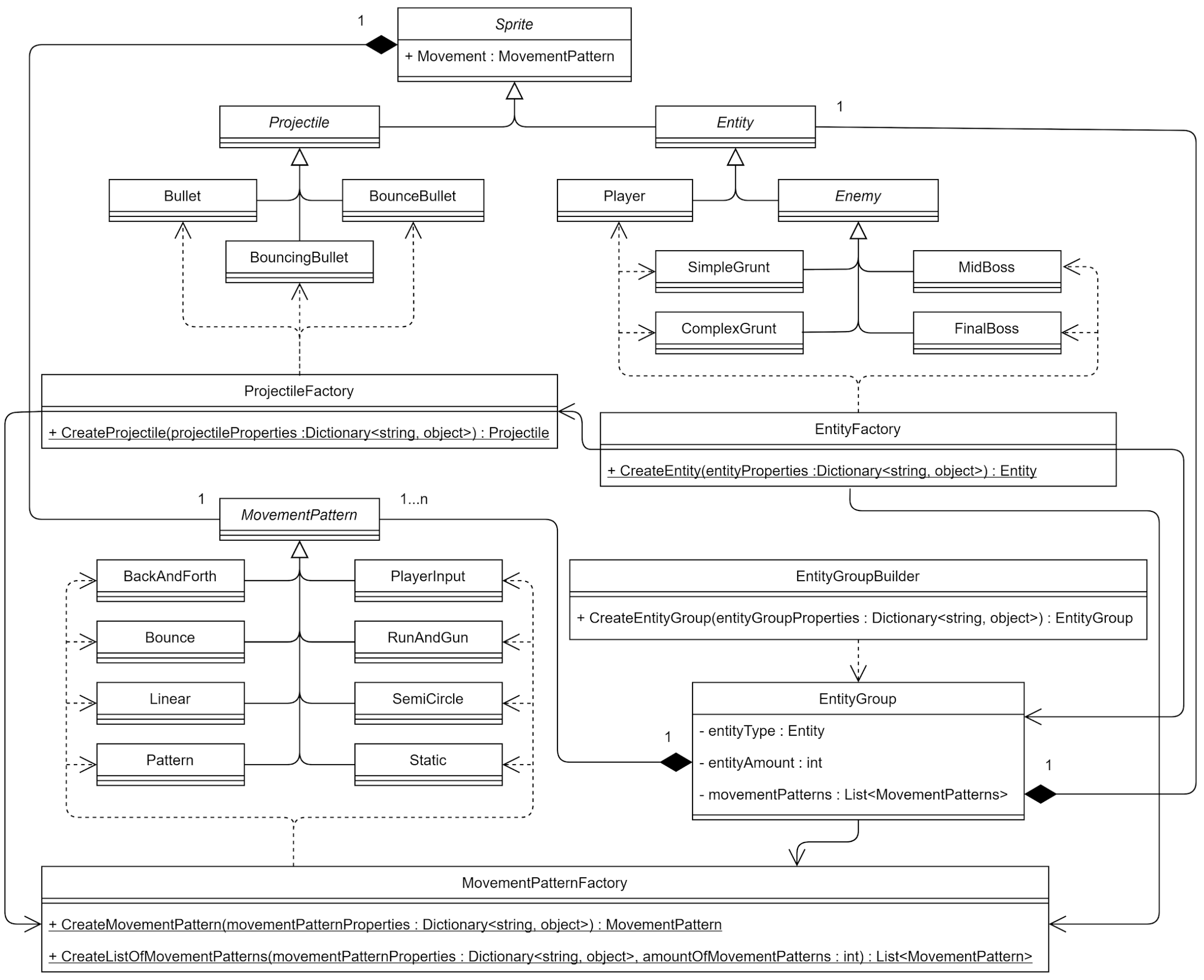
Sprite
+ Movement : MovementPattern
Projectile
Entity
Bullet
BounceBullet
BouncingBullet
Player
Enemy
SimpleGrunt
ComplexGrunt
MidBoss
FinalBoss
ProjectileFactory
+ CreateProjectile(projectileProperties :Dictionary<string, object>) : Projectile
EntityFactory
+ CreateEntity(entityProperties :Dictionary<string, object>) : Entity
MovementPattern
BackAndForth
PlayerInput
Bounce
RunAndGun
Linear
SemiCircle
Pattern
Static
EntityGroupBuilder
+ CreateEntityGroup(entityGroupProperties : Dictionary<string, object>) : EntityGroup
EntityGroup
- entityType : Entity
- entityAmount : int
- movementPatterns : List<MovementPatterns>
MovementPatternFactory
+ CreateMovementPattern(movementPatternProperties : Dictionary<string, object>) : MovementPattern
+ CreateListOfMovementPatterns(movementPatternProperties : Dictionary<string, object>, amountOfMovementPatterns : int) : List<MovementPattern>
1
1
1
1...n
1
1

Figure 2-9: Factory Method Pattern

2.3.2 Composite Pattern for Entities and Projectiles

Composite Pattern Mapping:

- Client: GameState
- Component: Sprite
- Leaves: Bullet, BounceBullet, and BouncingBullet
- Composite: Entity, Projectile
- Operation: Update
- Add: Attack

The GameState contains many sprites, which are either Projectiles or Entities. The status of Sprites in the game is handled by calls to Update. Entities currently contain a Projectile, and Projectiles are cloned and added to the GameState's list of Sprites in the Attack method. Sprites are removed when isRemoved is true on a Sprite, but this is done by the GameState. Projectiles are also a composite as they keep track of their parent Sprite as well, but sort of a leaf as well because Entities also contain a Projectile. This differs from the composite pattern by having both Entity and Projectile use a form of composition, and in Projectile having additional classes beneath it.

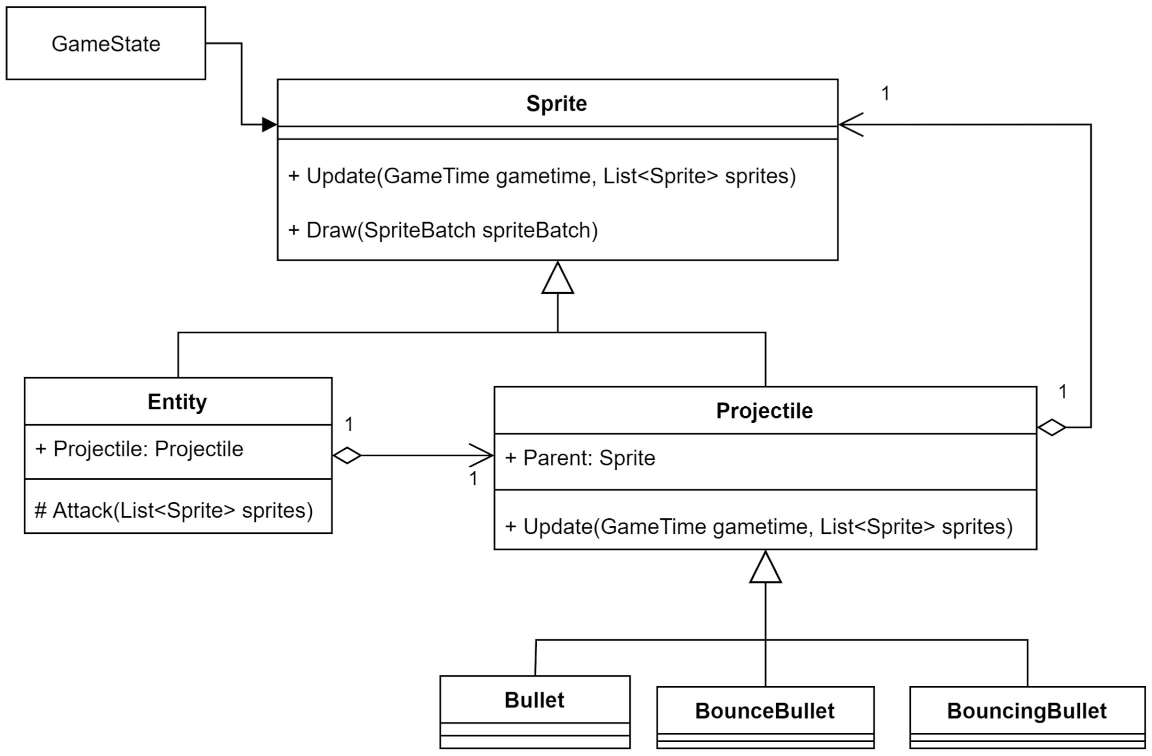
GameState
Sprite
+ Update(GameTime gametime, List<Sprite> sprites)
+ Draw(SpriteBatch spriteBatch)
1
Entity
+ Projectile: Projectile
# Attack(List<Sprite> sprites)
1
1
Projectile
+ Parent: Sprite
+ Update(GameTime gametime, List<Sprite> sprites)
1
Bullet
BounceBullet
BouncingBullet

Figure 2-10: Composite Pattern

2.3.3 Command Pattern for Collision Management

Command Pattern Mapping:

- Invoker: GameState
- Client: GameState
- Command: ICommand
- Receiver: Sprite
- ConcreteCommand: UpdateCommand, CollisionCheckCommand

The code differs from the design pattern slightly by having the GameState class function as both the invoker and client from the pattern. This was done due to concerns that adding another class solely for creating and executing two types of commands was unnecessary clutter, and would do more harm than good in this case.

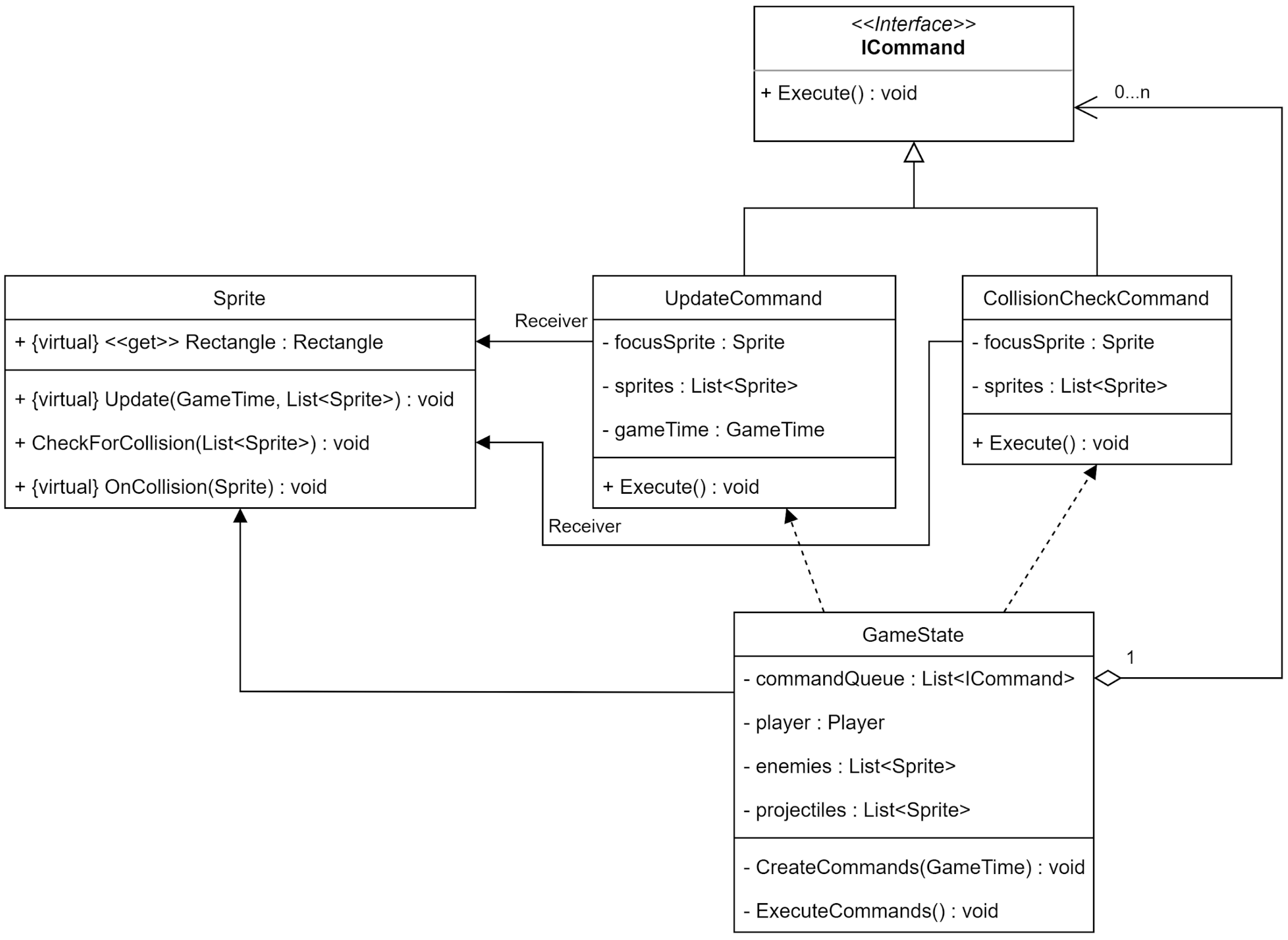

Figure 2-11: Command Pattern Class Diagram. Only attributes and operations in some way relevant to the commands have been included for the Sprite and GameState classes.

3. Subsystem Services

Figure 2-5 shows the architectural layout of this project's system, as well as which subsystems belong to each layer. Within the Presentation layer is the GUI. This subsystem handles the drawing of entities on the screen and runs the different states such as the menu states and the game state. Within the logic layer are the State Management and Sprite Management subsystems. The State Management subsystem informs the GUI with what states need to be run through StateChanger and also informs it with game related updates through GameManager. The Sprite Management subsystem supports the State Management with the CollisionChecker which handles entity collision and with SpriteUpdater which handles updates connected to sprite interaction. In the Data layer is the Game Setup subsystem which also supports the State Management. Game Setup provides the State Management with the WaveCreator service, which has information on wave generation, and the GameDictionaryLoader which handles loading from the game dictionary.

3.1 GUI Services

The GUI for this bullet hell game is responsible for the rendering of all game contents and graphical logic, such as screen size and managing MonoGame texture resources. The GUI component works closely with the StateManagement system, requiring both of its services to manage the game logic and switch between states, such as the various menu states and the game state itself. Overall, the GUI provides the actual Game service to the user, allowing them to interact with the logic through the graphical interfacing components.

Component	**Required Services**	**Provided Services**
GUI	● GameManager ● StateChanger	Game

Table 3-1: GUI Interface Required and Provided Services

3.2 Game Setup Services

The GameSetup service provides the ability to load game data from a JSON file. The game setup services are composed of two parts: the JSON file parsing and the game setup based on the parsed JSON file. The GameDictionaryLoader service provides a way to translate a JSON file into a game dictionary that will then be used in the creation of game objects such as waves, entity groups, entities, and so on. The WaveCreator provides the wave objects to the State Management at load time. The created waves are based on the GameDictionaryLoader service's dictionary. The waves are then used and rendered by the GameState to "play" the game.

Component	**Required Services**	**Provided Services**
Game Setup	None	● GameDictionaryLoader ● WaveCreator

Table 3-2: Game Setup Required and Provided Operations

3.3 Sprite Management Services

The CollisionChecker service provides the game with a way to check if sprites (such as projectiles, enemies, or the player) have collided with each other in the game space. The SpriteUpdater service provides a way to interact with sprites to update them in a way that can also be done within a command queue to be executed during the game loop. Both of the previous services require the use of the GUI's GameUpdater service. EntityCreator supplies methods for creating different types of entities and their related components, requiring the GameSetup component's GameDictionaryInfo service to do so.

Component	**Required Services**	**Provided Services**
Sprite Management	None	● CollisionChecker ● SpriteUpdater

Table 3-3: Sprite Management Required and Provided Services

3.4 State Management Services

GameManager provides the GUI with constantly updated information such as how many lives the player has and where sprites should be drawn. The StateChanger service is responsible for switching between the various game states, such as Select Difficulty, Game Over Win, Game Over Lose, Rebind Keys, Menu, Options, and the Game State itself. Since the State Management component is where the bulk of the game logic is executed, it requires the use of services from both the Game Setup and Sprite Management components.

Component	**Required Services**	**Provided Services**
State Management	● WaveCreator ● GameDictionaryLoader ● CollisionChecker ● SpriteUpdater	● GameManager ● StateChanger

Table 3-3: State Management Required and Provided Services