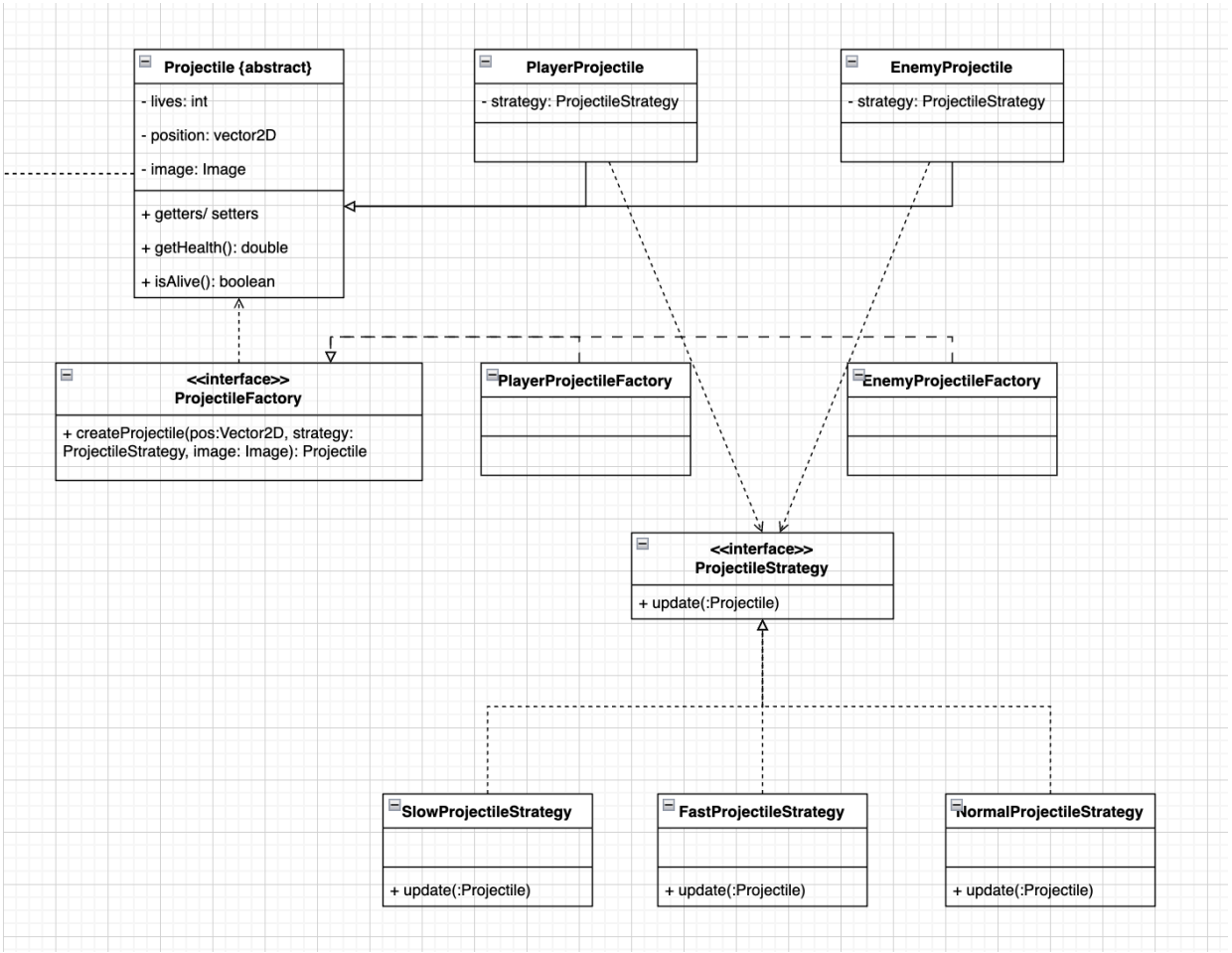
1. Code review

Overall, the design of the given codebase mostly follows good OOP principles. We can review this on SOLID principles:

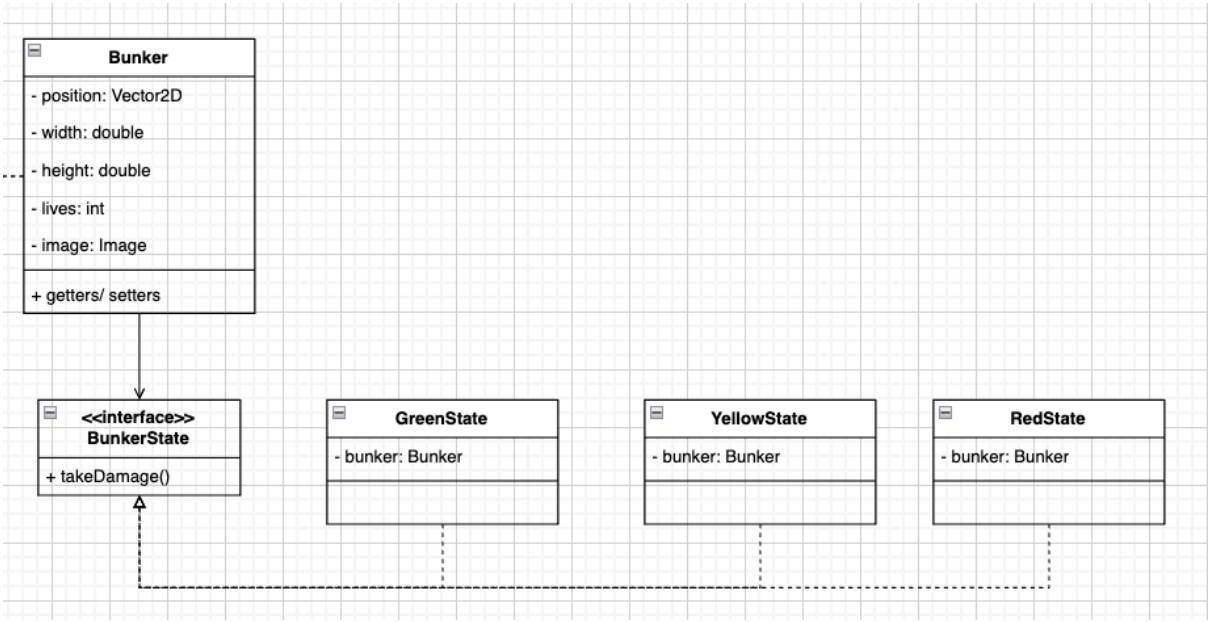
- Single Responsibility Principle (SRP): Most classes have a clear and single responsibility. But there is one exception in the Renderable class, where I think we can separate the health and state of the object in another interface.
- Open-Closed Principle (OCP): The code is really open for extension.
- Liskov Substitution Principle (LSP): All of the subclasses can be used interchangeably with their base classes.
- Interface Segregation Principle (ISP): Interfaces in the codebase are focused.
- Dependency Inversion Principle (DIP): High-level modules depend on abstractions.

Despite that, the codebase has 2 redundant classes that I think were meant to be used to handle collision detection: Collider interface and Box collider class.
The design patterns used in the codebase helped a lot in achieving good OOP principles. These patterns were the patterns I was required to implement in Assignment 2, and I have discussed their advantages and disadvantages in that assignment report.
To sum up the use of design patterns, I have created an UML class diagram for the initial codebase:

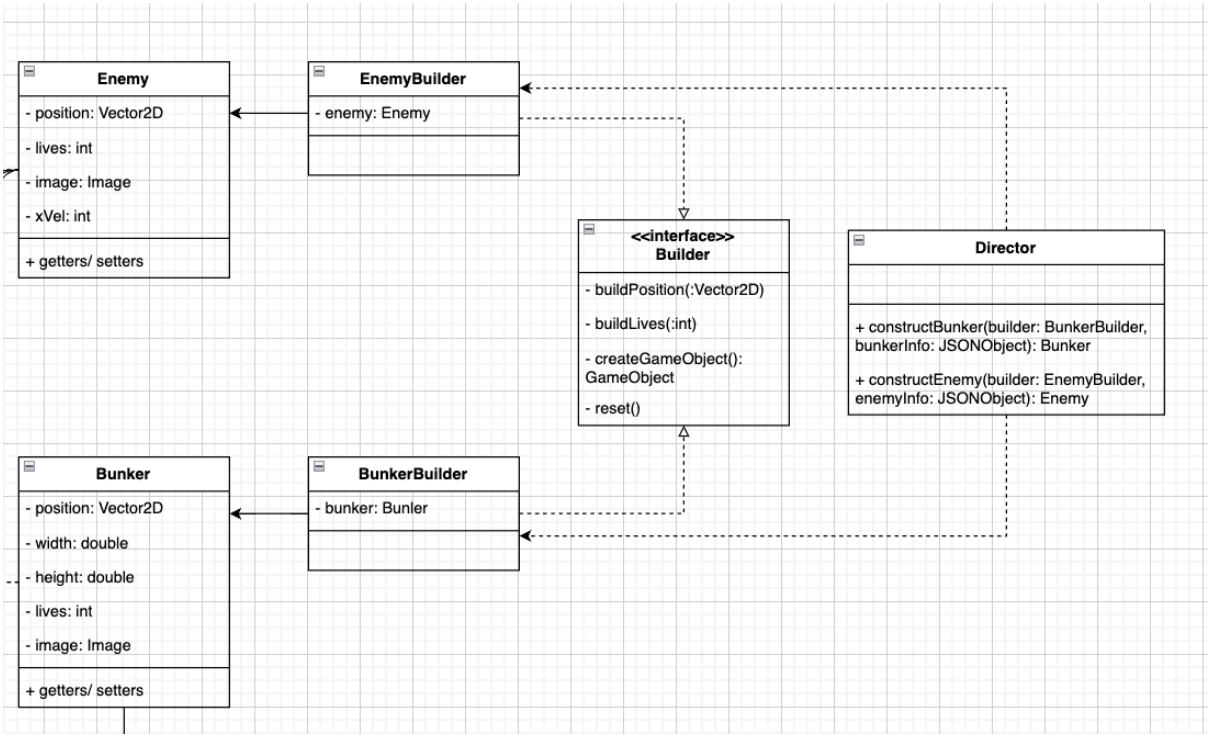
- The Factory method pattern is used to create a Projectile. And a projectile's movement is decided using the Strategy pattern:



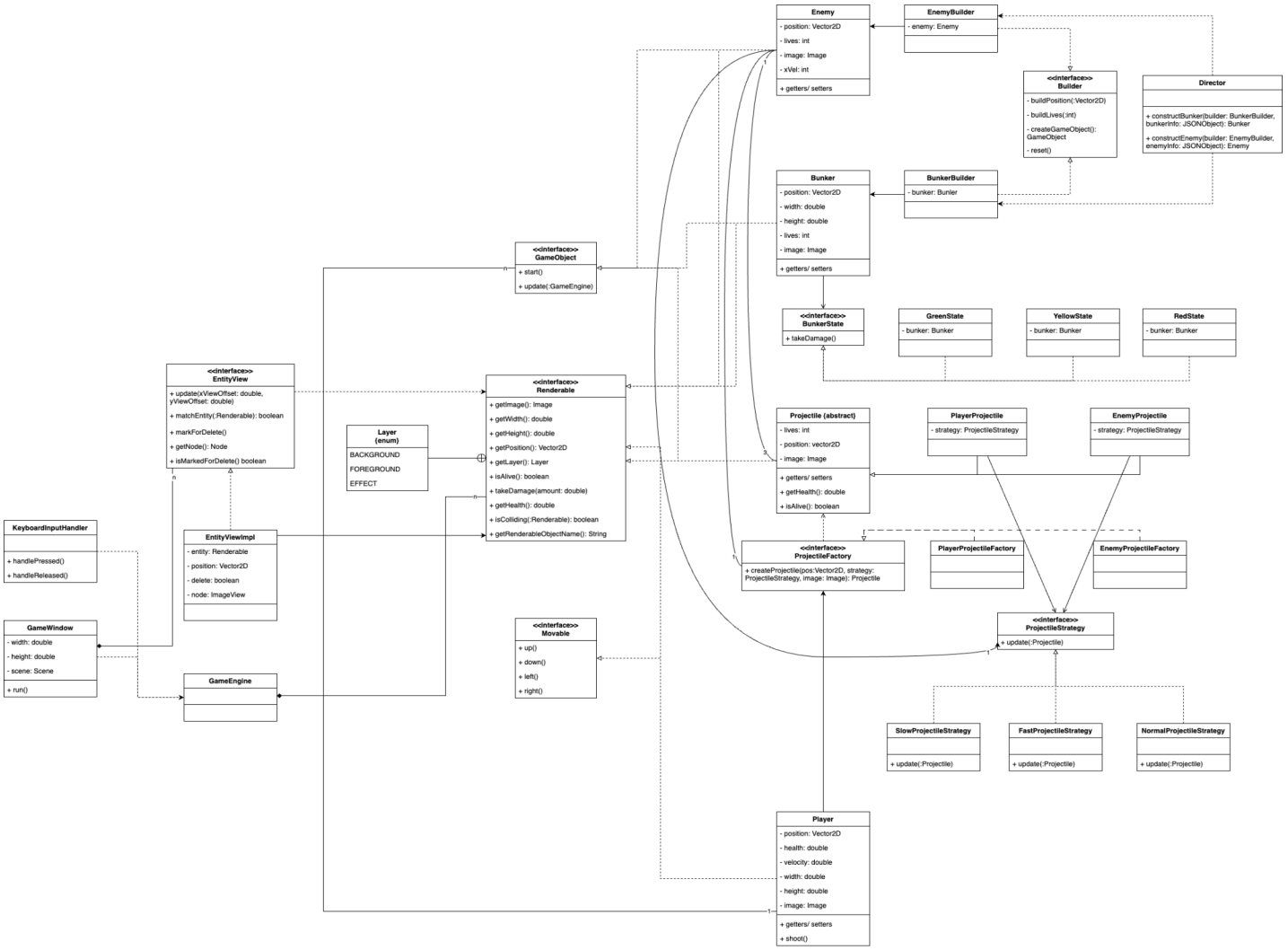
- A bunker's internal state is controlled by the state pattern



- Bunker and Enemy is created using the Builder pattern:



The Class diagram for the codebase:



Although the codebase design is good, it provided little to no comments, so it's kind of hard to follow the function flow and I often have to read the implementation and follow the function call to understand the program's flow.
On first review, the codebase's design allows for easy extension since the classes are well modularized and the codebase achieves low coupling and high cohesion.

2. Feature extensions

2.1. Difficulty picking

To implement difficulty picking, I chose to implement a class called DifficultMenu. On startup, instead of immediately loading into the game, the App loads into the scene of this Class. This class is used to define an User Interface for the player to pick the difficulty so that the Game can load the corresponding Json. I don't use any design pattern to achieve this feature.
Adding this feature required me make the following changes to the codebase:

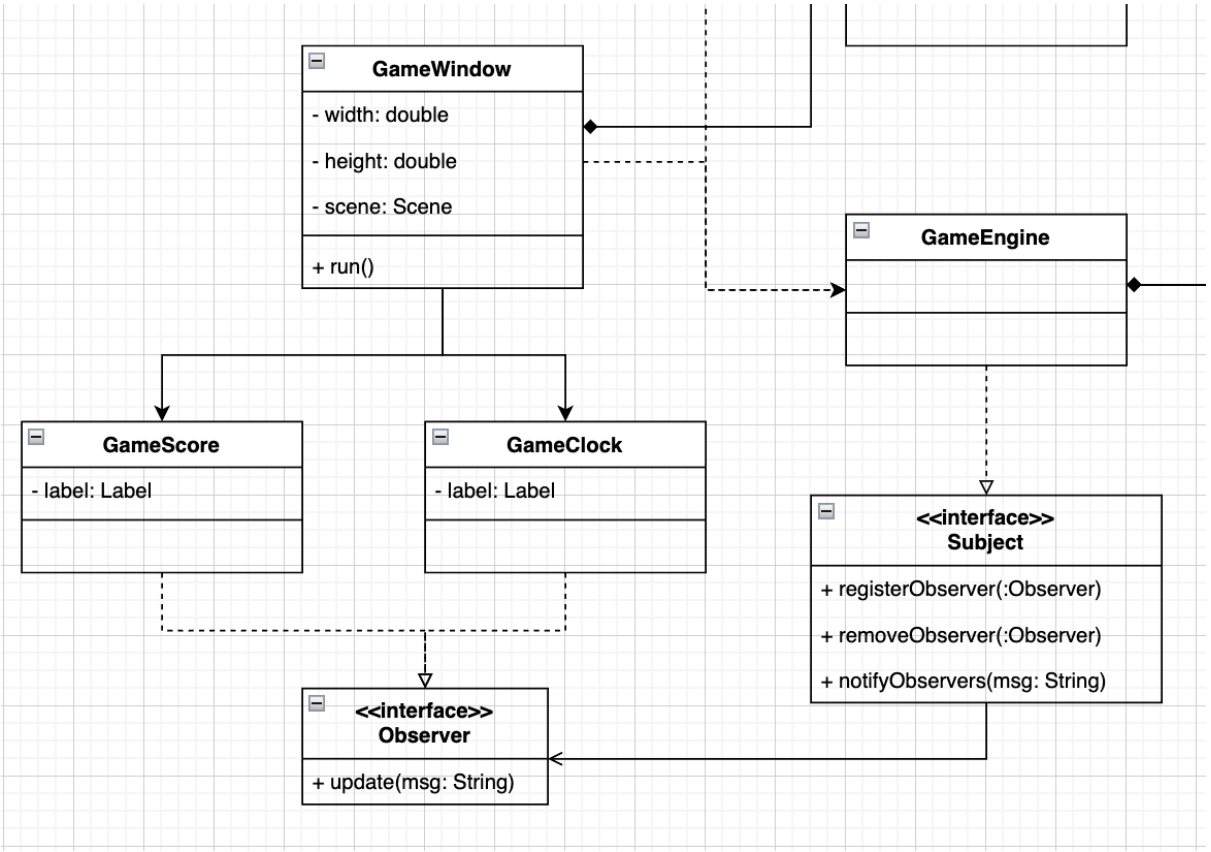
- Change the implementation of App class: I needed to change how the program starts so that it loads into difficulty picking menu instead of immediately loading into the game.
- Add a new file/class called DiffPicker to represent the difficulty picking menu.

2.2. Time and score

To implement the functionality to display time and score, I used the Observer pattern. My idea is to create observers to observe the state of the current game, then use that state to update the displaying time and score.
To achieve this functionality, I have to make the following changes to the codebase:

- Add 2 new interfaces for the pattern: Subject and Observer. These 2 interfaces are the base to implement the pattern.
- Make GameEngine class implement the Subject interface so that other classes can get notified from its internal state. These changes added 3 method overwrites to the class's body and added some function calls to existing method's body to notify the Observers when necessary.
- I also added a Timeline object inside the GameEngine class to act as a timer, this timeline is created in the class's constructor.

- Creating 2 new classes to represent the GameClock and GameScore. These 2 classes both implement the Observer interface, each class has a Label object to represent the displayed Label.
- Addition to the GameWindow class so that it can create and display the GameClock and GameScore's Label-s.

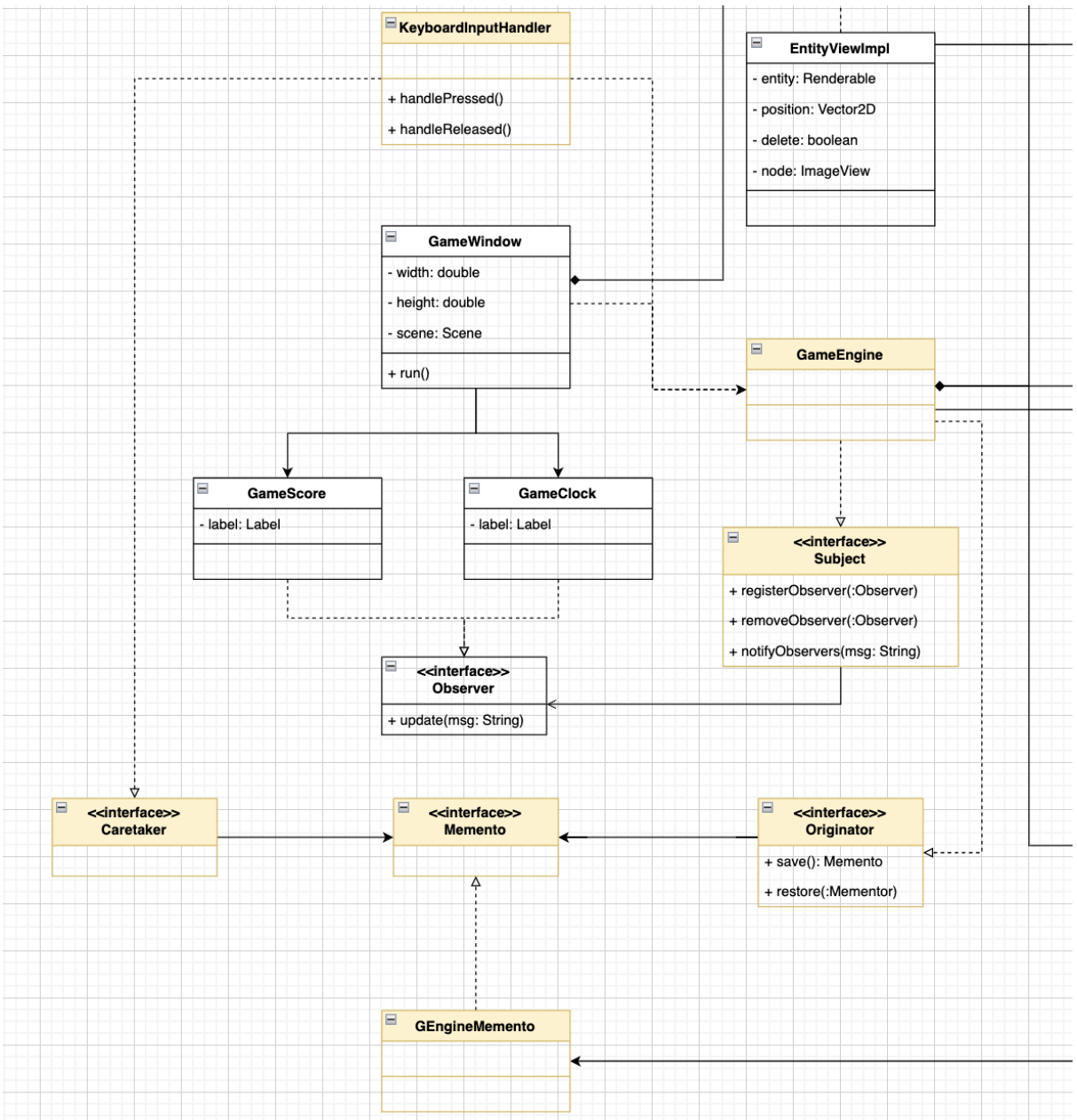


2.3. Undo and Cheat

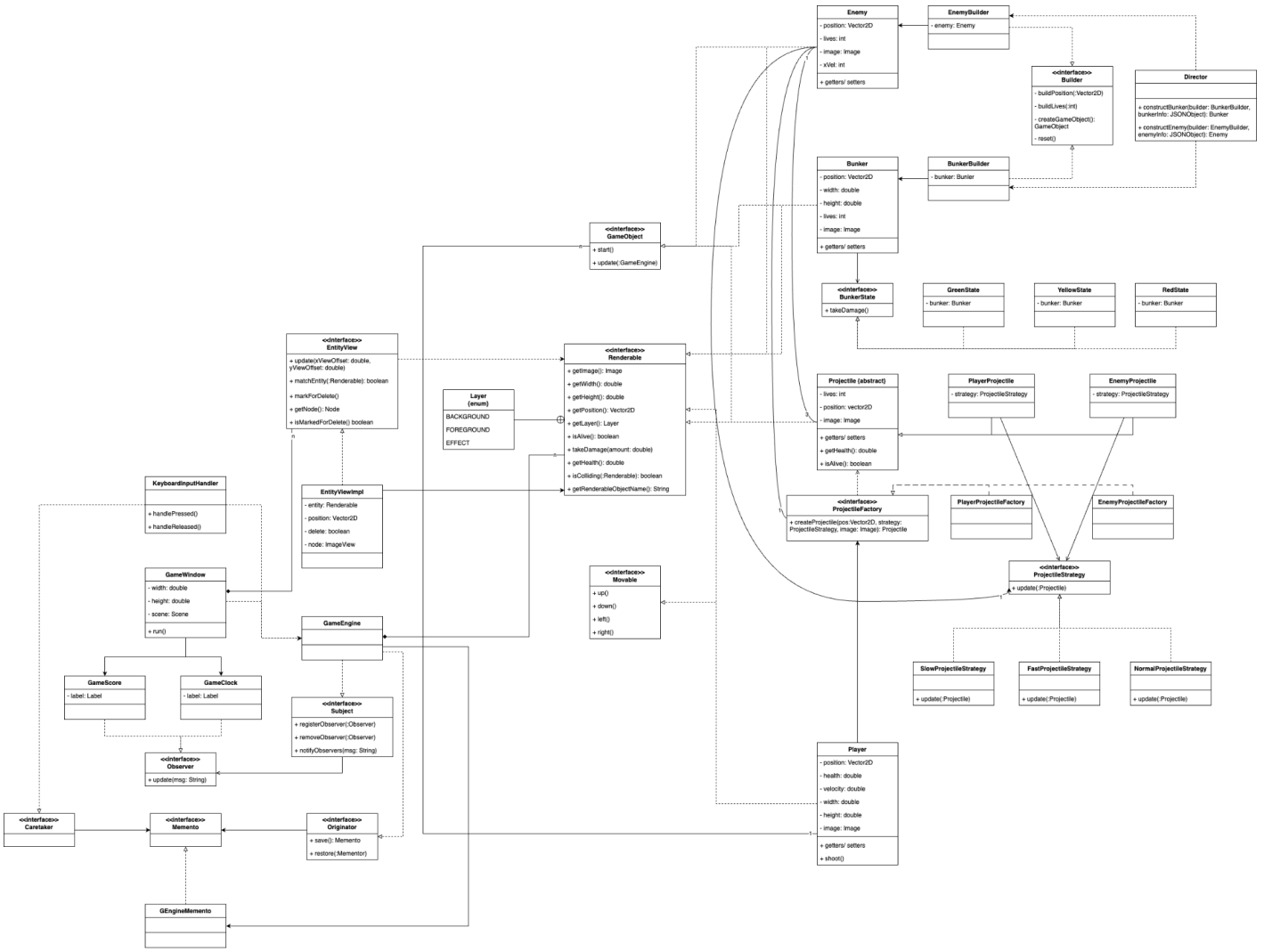
To implement these functionality, I had to use two design patterns: Prototype and Memento. The idea is that the Prototype pattern can help in creating a deep copy of the current game state. I can then use that copy whenever I want to undo using the Memento pattern.

The implementation requires the following changes to the codebase:

- Add the Prototype interface for the prototype pattern.
- Changes to Enemy, Bunker, Projectile, Player to implement Prototype interface.
- Add Memento, Originator and Caretaker interface for the Memento pattern.
- Add GEngineMemento class, implement Memento as the memento for a GameEngine state.
- Changes to GameEngine class to make it implements the Originator interface.
- Changes to KeyboardInputHandler to implements the Caretaker interface.


KeyboardInputHandler
+ handlePressed()
+ handleReleased()
EntityViewImpl
- entity: Renderable
- position: Vector2D
- delete: boolean
- node: ImageView
GameWindow
- width: double
- height: double
- scene: Scene
+ run()
GameEngine
GameScore
- label: Label
GameClock
- label: Label
<<interface>>
Subject
+ registerObserver(:Observer)
+ removeObserver(:Observer)
+ notifyObservers(msg: String)
<<interface>>
Observer
+ update(msg: String)
<<interface>>
Caretaker
<<interface>>
Memento
<<interface>>
Originator
+ save(): Memento
+ restore(:Mementor)
GEngineMemento

Final UML:


Enemy
- position: Vector2D
- lives: int
- image: Image
- xVel: int
+ getters/ setters
EnemyBuilder
- enemy: Enemy
<<interface>>
Builder
- buildPosition(:Vector2D)
- buildLives(:int)
- createGameObject(): GameObject
- reset()
Director
+ constructBunker(builder: BunkerBuilder, bunkerInfo: JSONObject): Bunker
+ constructEnemy(builder: EnemyBuilder, enemyInfo: JSONObject): Enemy
Bunker
- position: Vector2D
- width: double
- height: double
- lives: int
- image: Image
+ getters/ setters
BunkerBuilder
- bunker: Bunker
<<interface>>
GameObject
+ start()
+ update(:GameEngine)
<<interface>>
BunkerState
+ takeDamage()
GreenState
- bunker: Bunker
YellowState
- bunker: Bunker
RedState
- bunker: Bunker
<<interface>>
EntityView
+ update(xViewOffset: double, yViewOffset: double)
+ matchEntity(:Renderable): boolean
+ markForDelete()
+ getNode(): Node
+ isMarkedForDelete(): boolean
<<interface>>
Renderable
+ getImage(): Image
+ getWidth(): double
+ getHeight(): double
+ getPosition(): Vector2D
+ getLayer(): Layer
+ isAlive(): boolean
+ takeDamage(amount: double)
+ getHealth(): double
+ isColliding(:Renderable): boolean
+ getRenderableObjectName(): String
Layer
(enum)
BACKGROUND
FOREGROUND
EFFECT
Projectile (abstract)
- lives: int
- position: vector2D
- image: Image
+ getters/ setters
+ getHealth(): double
+ isAlive(): boolean
PlayerProjectile
- strategy: ProjectileStrategy
EnemyProjectile
- strategy: ProjectileStrategy
KeyboardInputHandler
+ handlePressed()
+ handleReleased()
EntityViewImpl
- entity: Renderable
- position: Vector2D
- delete: boolean
- node: ImageView
<<interface>>
ProjectileFactory
+ createProjectile(pos:Vector2D, strategy: ProjectileStrategy, image: Image): Projectile
PlayerProjectileFactory
EnemyProjectileFactory
GameWindow
- width: double
- height: double
- scene: Scene
+ run()
<<interface>>
Movable
+ up()
+ down()
+ left()
+ right()
<<interface>>
ProjectileStrategy
+ update(:Projectile)
GameEngine
SlowProjectileStrategy
+ update(:Projectile)
FastProjectileStrategy
+ update(:Projectile)
NormalProjectileStrategy
+ update(:Projectile)
GameScore
- label: Label
GameClock
- label: Label
<<interface>>
Subject
+ registerObserver(:Observer)
+ removeObserver(:Observer)
+ notifyObservers(msg: String)
<<interface>>
Observer
+ update(msg: String)
Player
- position: Vector2D
- health: double
- velocity: double
- width: double
- height: double
- image: Image
+ getters/ setters
+ shoot()
<<interface>>
Caretaker
<<interface>>
Memento
<<interface>>
Originator
+ save(): Memento
+ restore(:Memento)
GEngineMemento