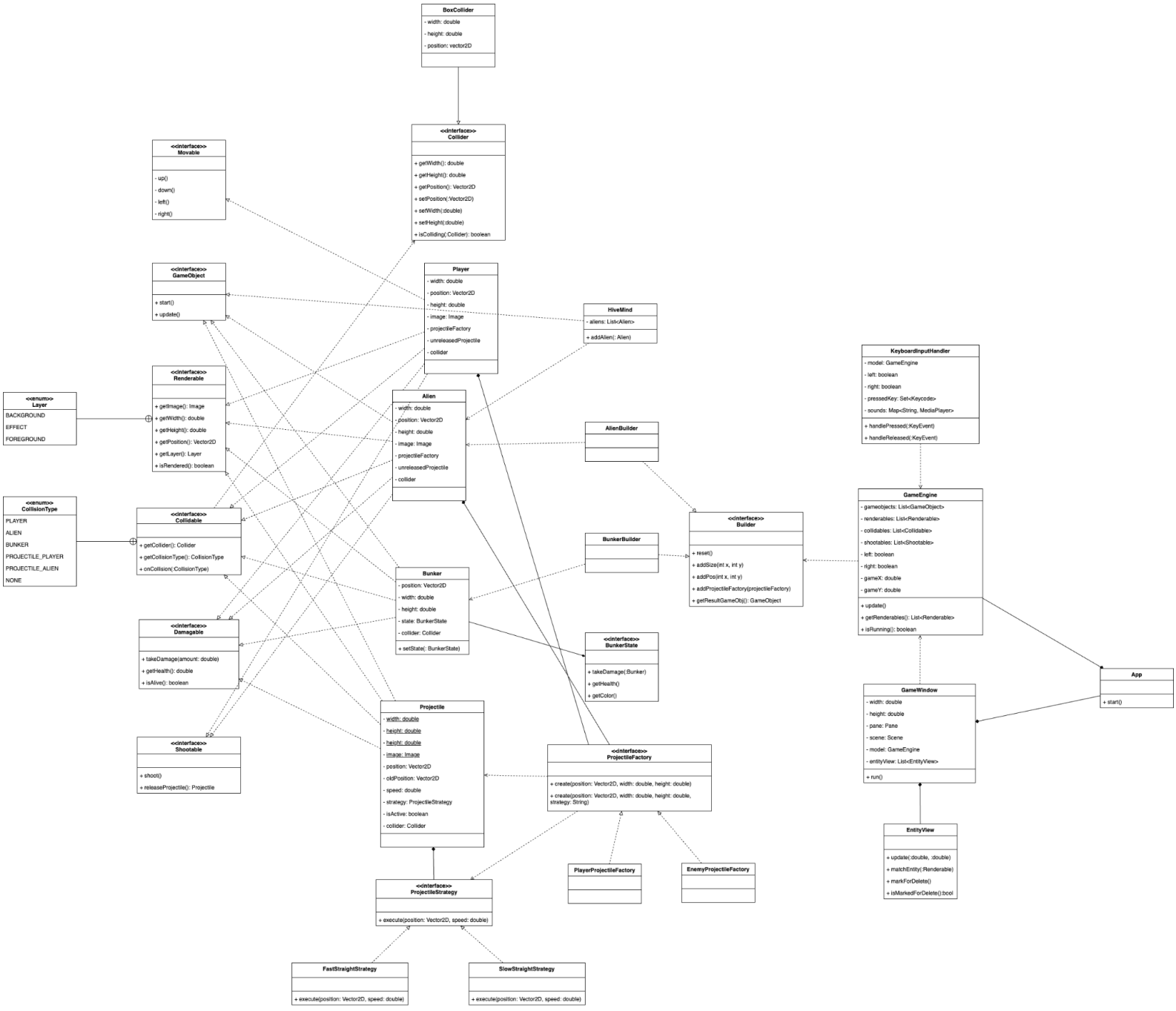
1. New UML Class diagram



2. Changes from assignment 1 UML

The design I created for assignment 1 gave me an overall view for the application, but it didn't use any design pattern and hence had a lot of flaws

in terms of SOLID/GRASP principles. For the assignment 2's design, I was asked to use some design patterns, and I think by using these patterns, the design was improved while still retaining the core ideas of my original design.

In addition to adding new interfaces/classes for implementing the required design patterns, I also changed the names and details of some classes from my original design to better fit the requirements and provided codebase. I changed those because, as mentioned above, there are flaws in the original design and the codebase suggested some ways to fix those flaws, so I took the suggestion and changed the design.

3. Design patterns

3.1. Factory method pattern

In terms of SOLID principle: the Factory Method Pattern helps in achieving the SRP by separating the responsibility of object creation from the client code that uses the objects. This promotes a cleaner separation of concerns. The pattern supports OCP by allowing you to introduce new subclasses (new types of products) without modifying existing client code. This is because clients depend on the abstract Creator interface, not concrete product classes.

In terms of GRASP: the Factory Method Pattern adheres to the Creator principle in GRASP, as it defines a separate creator (factory) class responsible for creating objects. This promotes low coupling between the creator and the objects it creates. The Factory Method Pattern can also align with the Information Expert principle when the factory class has the necessary knowledge and context to create the appropriate objects.

This pattern was used in my code to help with the construction of a Projectile object. It helped to abstract the creation of the object and to allow further expansion of the system. Introducing the pattern into my design comes with a downside as it creates more complexity in the codebase.

3.2. State pattern

The State Pattern is a powerful design pattern that promotes clean, maintainable code by encapsulating the behavior associated with different states and allowing objects to transition between states dynamically.

In terms of SOLID: The State Pattern can help adhere to SRP by encapsulating the behavior associated with different states in separate state classes, it supports OCP by allowing you to add new states (state classes) without modifying the context, it also promotes adherence to LSP because each state class is a subtype of a common state interface or abstract class.

In terms of GRASP: The State Pattern typically assigns the responsibility for managing the behavior associated with a state to the state object itself. This aligns with the Information Expert principle.

In my code, it was used to represent the changing between different states of a Bunker object. This helped to separate the state of a Bunker from the object itself. The drawback of this pattern is that it introduces more complexities into the codebase and there are some repeated code segments among the states.

3.3. Builder pattern

In terms of SOLID: The Builder Pattern aligns with SRP by separating the construction of complex objects from their representation. The builder class is responsible for constructing the object, and the constructed object itself is responsible for its own behavior. This separation promotes a single responsibility for each class. The Builder Pattern supports OCP by allowing you to create new builders for different types of objects without modifying the client code. Clients interact with the builder interface, not with concrete builders, making it easy to extend or add new builders without changing existing code.

In term of GRASP:

- Creator: In the Builder Pattern, the builder object is responsible for creating and configuring the complex object. This aligns with the Creator principle in GRASP, as the creation logic is encapsulated within a separate builder class.
- Information Expert: The builder object is typically designed to have knowledge of the construction process and the details of the object being built. This aligns with the Information Expert principle as the builder is an expert in constructing the complex object.
- Controller: Depending on the design, a director or controller object can be used to orchestrate the construction process by using a builder. This separation of responsibilities aligns with the Controller principle in GRASP, as it helps maintain a clean separation of concerns between constructing objects and using them.
- High Cohesion: The Builder Pattern encourages high cohesion by grouping the construction-related operations within a dedicated builder class. This ensures that all operations related to object construction are in one place, making the code more organized and maintainable.

In my code, the pattern is used to help with the construction of an Enemy object and Bunker object. It helped to simplify the construction of the object and allows flexible creation of that object.

3.4. Strategy pattern

In terms of SOLID: The Strategy Pattern promotes adherence to SRP by encapsulating each algorithm in a separate strategy class. Each strategy has a single responsibility: to implement a specific algorithm. This separation of concerns ensures that each class has only one reason to change. It supports OCP by allowing you to add new strategies (algorithm implementations) without modifying existing client code. Clients interact with the strategy

interface or abstract class, not with concrete strategies. This makes it easy to extend or replace strategies as needed.

In terms of GRASP: as each strategy class has the knowledge and expertise required to perform its particular task, it aligns with the Information Expert principle.

In the code, it was used to represent different strategies of a Projectile. This helped separate the strategy from the Projectile, which can help in further addition of strategies.