Design Patterns

Factory Method: 3D Object Creation

Description

The factory method has been applied to all 3D Object used in our scene.This includes the Ball3D, Cue3D, and Table3D classes.

Why

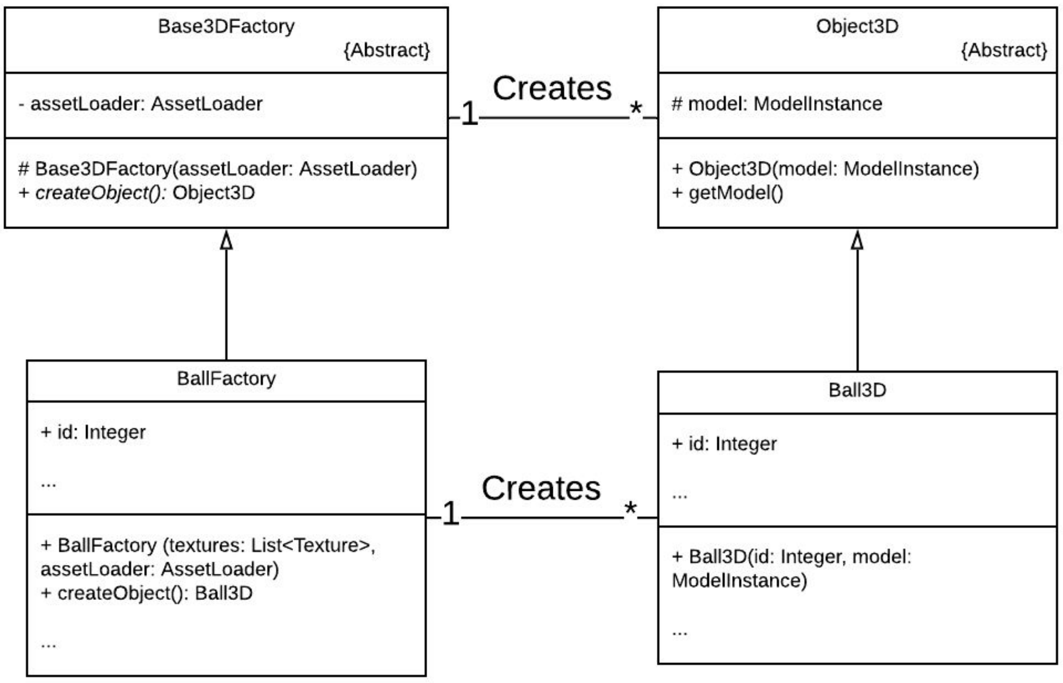
Applying this pattern allowed us to:

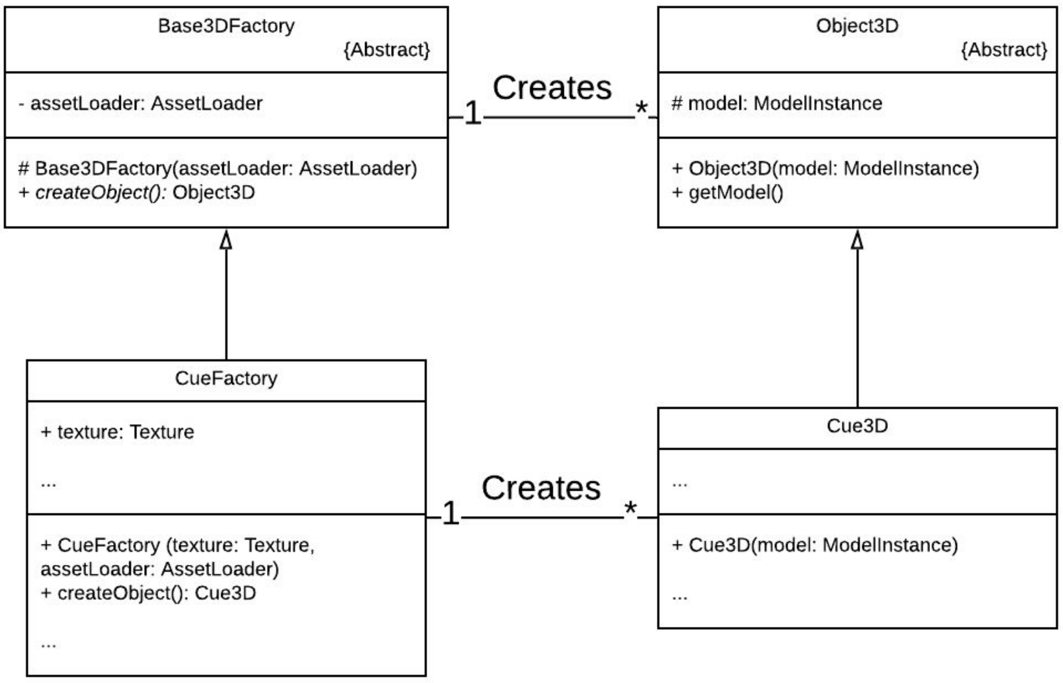
- Have the logic for creating the specific objects - which was quite complicated at times - separated from each other. This helped code understandability and maintainability a tremendous amount.
- Improves testability: Since initialization of objects is now moved to separate classes, we no longer need to care about object initialization when testing the objects itself, increasing ease of test setting up & overall testability as compared to keeping the initialization logic in those objects.
- Generalization of 3D element creation: The implementation of the Factory Method allowed us to generalize the creation of objects in the 3D element family, which allows for greater levels of abstraction and re-use. If we were ever to swap out some 3D Game Elements in other classes, this is now far easier to achieve with the current class structure.

How

- An abstract class has been created to generalize 3D objects. This holds functionality common to all 3D game elements, namely getting coordinates and getting the model instance. In the slides, the factory method design pattern is portrayed to use an interface to generalize the Product. However, we chose to use an abstract class for this purpose, so that we could move some functional implementation into this class.
- The concrete classes, like Ball3D, have been modified to extend this abstract class.
- An abstract class has been created to generalize the factory classes. The concrete factories extend this abstract class.
- A parallel class structure is formed for the 3D objects and factories. A BallFactory creates Ball3D, a CueFactory creates Cue3D and so on.

Diagram


Base3DFactory
{Abstract}
- assetLoader: AssetLoader
# Base3DFactory(assetLoader: AssetLoader)
+ createObject(): Object3D
Creates
1
*
Object3D
{Abstract}
# model: ModelInstance
+ Object3D(model: ModelInstance)
+ getModel()
BallFactory
+ id: Integer
...
+ BallFactory (textures: List<Texture>, assetLoader: AssetLoader)
+ createObject(): Ball3D
...
Creates
1
*
Ball3D
+ id: Integer
...
+ Ball3D(id: Integer, model: ModelInstance)
...


Base3DFactory
{Abstract}
- assetLoader: AssetLoader
# Base3DFactory(assetLoader: AssetLoader)
+ createObject(): Object3D
Creates
1
*
Object3D
{Abstract}
# model: ModelInstance
+ Object3D(model: ModelInstance)
+ getModel()
CueFactory
+ texture: Texture
...
+ CueFactory (texture: Texture, assetLoader: AssetLoader)
+ createObject(): Cue3D
...
Creates
1
*
Cue3D
...
+ Cue3D(model: ModelInstance)
...

In code

The corresponding creator classes can be found under the **com.sem.pool.factories** package:

- Base3DFactory.java
- BallFactory.java
- CueFactory.java
- TableFactory.Java

The corresponding element classes can be found under the **com.sem.pool.scene** package:

- Ball3D.class
- CueBall3D.class
- EightBall3D.class
- RegularBall3D.class
- Table3D.class
- Cue3D.class

Observer Pattern: Observable Game class

Description

The observer pattern has been implemented primarily between two classes: Game and the Game State, with support for multiple extensions of the Game & Game State class (interfaces were used).

Context

Game class - handles Game loop logic: moving balls, triggering collisions (ball potting, ball vs board collisions), accessing Game State and making decisions based on the current Game State.

Game State class - handles inner Game logic: assigning potted balls to Players, deciding on a winner at the end of the Game, assigning ball types to players, keeping track of current turn and other game variables.

Why

The Observer pattern has been implemented for the following reasons:

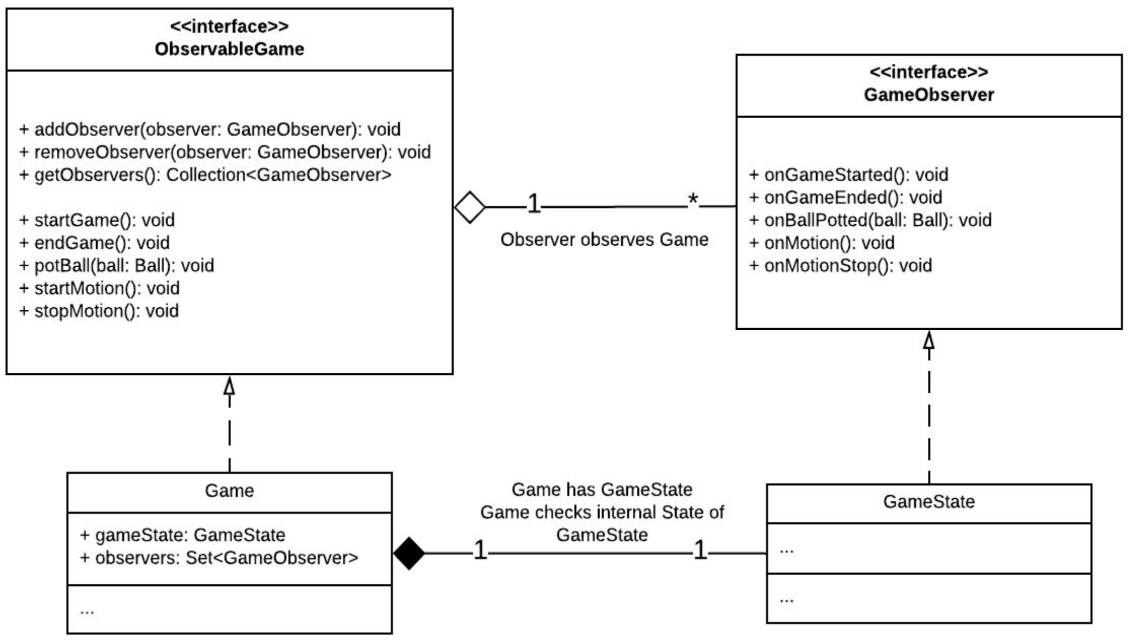
- Changes in Game State variables rely on the progression of the Game loop, as well as the Game class itself. Since the Game class handles the Game loop, it makes sense that the Game class would propagate loop events to the Game State class where it's due so that the Game State class knows what occurred in the Game loop so that it can process it further.
- With the pattern in place, we can be very flexible in swapping out the current Game or Game State classes. For example, if we want to devise a new Pool game mode (e.g. single-player practice mode), all we would need to do is swap out the Game State which corresponds to the mode, and thus we make such extensions far easier.
- There is one catch to the pattern: What we did is add some direct coupling between the Game & Game State (the composition), despite the original Observer pattern not having such a relation. This is because the Game object needs to get data from the Game State to make decisions (specifically, the internal state, which was modelled in the class diagram of assignment 2). However, it is important to note that we implemented this pattern with more extensibility in mind. If we were ever to extend the Game to have more complex GUI, which shows potted balls for example, it would make sense to make the GUI component an observer of the Game object so that it can receive all the events, and update the GUI according to the current Game state.

How

- An interface ObservableGame has been created to represent Games that can be observed; This interface holds methods relevant for adding, removing and retrieving observers, and methods for triggering events to the observers.

- An interface GameObserver has been made, which represents an Observer of the Game. The interface contains methods to react to the events sent by the Game.
- The concrete classes Game & GameState (which already existed) have been adapted to implement these interfaces. Some methods in Game and GameState were renamed to match the interface names, while some had to be added.

Diagram



Code Location

The corresponding classes can be found under the **com.sem.pool.game** package:

- GameState.java
- Game.java
- GameObserver.java
- ObservableGame.Java

Template Method: Pool Ball Types

Description

The template method has been applied to form a class hierarchy for Pool Balls, one for each type: Cue Ball, 8-Ball, Regular Ball

Why

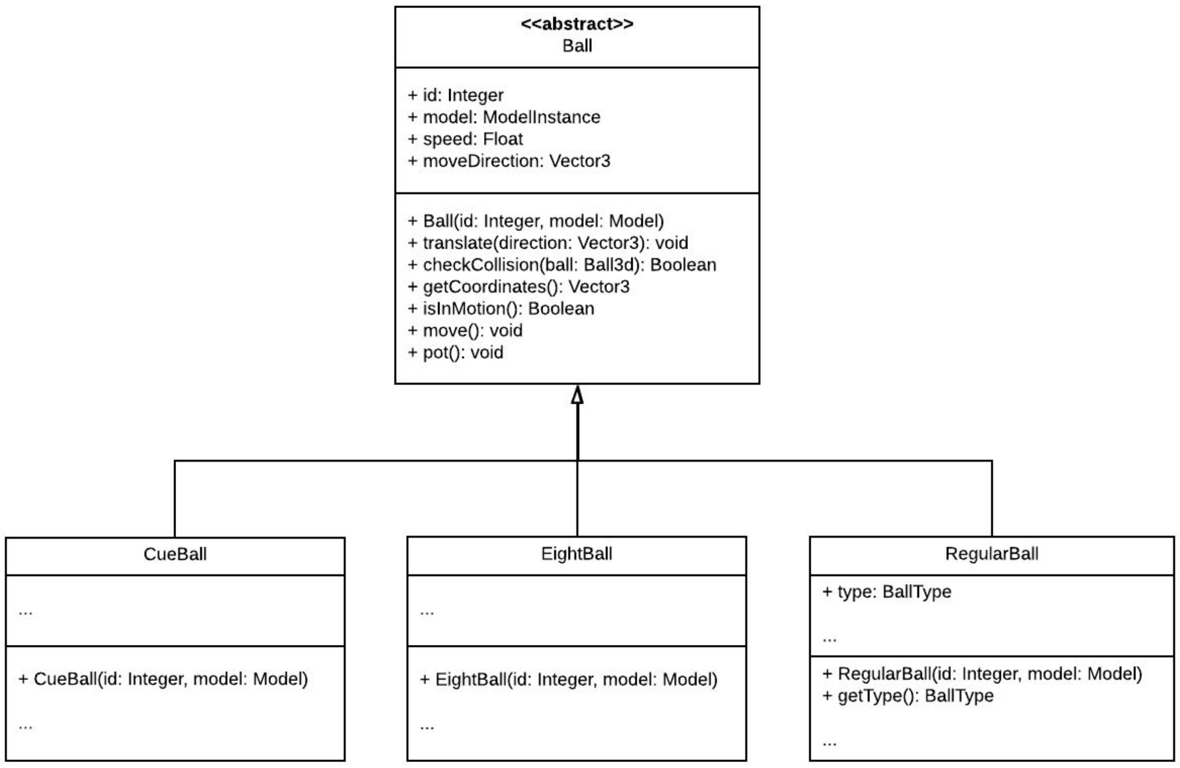
The pattern has been implemented to allow for flexibility of handling different Pool Ball types. Due to certain game rules depending on ball types (e.g. potting cue ball results in loss of turn, or potting 8-ball when all balls are potted results in a victory), we found it quite important to structure the pool ball system in a way that avoids complex logic by separating the classes. The alternative would have been distinguishing classes by ID, but this would have made the code more complex and far less flexible. We chose the Template method specifically since there is a lot of shared functionality between all the different ball types.

Moreover, if we ever wanted to make changes per Ball type (perhaps some special Pool game extension, such as making the 8-ball be heavier than other balls), this would be far easier to accomplish with different classes.

How

- An abstract class was created for the Ball class
- Most of the common functionality, namely movement, physical potting, collision checking was implemented in the abstract Ball class.
    - Although the abstract class has an implementation for some methods, subclasses could override these methods to change the behaviour to something fitted for that subclass.
- Three specific classes have been created for each Ball type - Cue Ball, 8-ball and Regular Ball
- The Regular Ball class was extended with additional Ball Type property which distinguishes Striped from Solid balls
- The Eight Ball & Cue Ball were simply extensions of the standard abstract Ball class; They do not contain any new functionality, but simplify Ball type distinguishing by class type.

Diagram



<<abstract>>
Ball
+ id: Integer
+ model: ModelInstance
+ speed: Float
+ moveDirection: Vector3
+ Ball(id: Integer, model: Model)
+ translate(direction: Vector3): void
+ checkCollision(ball: Ball3d): Boolean
+ getCoordinates(): Vector3
+ isInMotion(): Boolean
+ move(): void
+ pot(): void
CueBall
...
+ CueBall(id: Integer, model: Model)
...
EightBall
...
+ EightBall(id: Integer, model: Model)
...
RegularBall
+ type: BallType
...
+ RegularBall(id: Integer, model: Model)
+ getType(): BallType
...

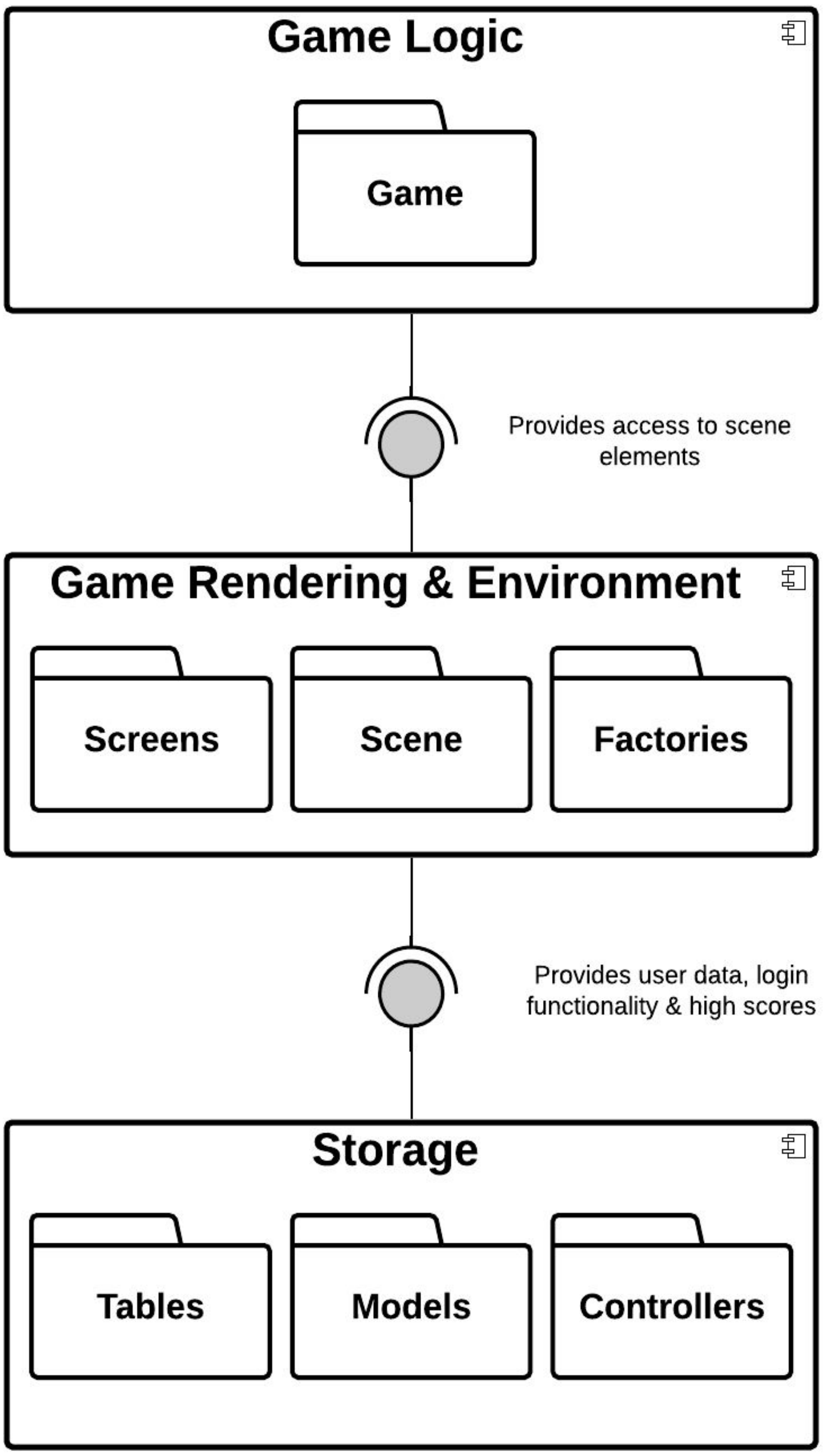
Code Location

The corresponding classes can be found under the **com.sem.pool.scene** package:

- Ball3D.java
- CueBall3D.java
- EightBall3D.java
- RegularBall3D.java

System Architecture: Layered Architecture

Diagram


Game Logic
Game
Provides access to scene elements
Game Rendering & Environment
Screens
Scene
Factories
Provides user data, login functionality & high scores
Storage
Tables
Models
Controllers

Component descriptions

Storage

The Storage component handles **data storage** in a database, and defines the structure of the data. The component's responsibility is to handle data creation, reading, updating and deletion (CRUD) of user and player data, namely credentials and game statistics. The component provides an interface to the Game Rendering & Environment component, which can use the Controllers to query for the data and get responses, without knowing the internal organization of the data.

The component contains 3 packages:

- Tables - contains relational schema definitions. Intended to be used primarily only by the component.
- Models - contains the models of the data entries, namely the User object containing credentials.
- Controllers - contains objects that allow to query the database for responses & updates to the data.

Game Rendering & Environment

The component handles menu and game **rendering**, scene **instantiation** and **controlling** the game environment. The main responsibility of the component is to present the game to the Player with all the relevant elements of the game, control the environment (e.g. handle collisions), and allow the user to navigate around the environment (input). The interface provided to the Game Logic includes the scene elements and input handling.

The component contains 3 packages:

- Screens - Screens of the game, both 2D menus and the 3D game screen. Handles rendering & UI. The package is intended to be used mainly by the component itself.
- Scene - Scene elements of the Game: Pool balls, Table, Cue, Camera. The elements have their own set of functionalities (e.g. for movement and collisions)
- Factories - Handles abstracting the instantiation of the scene elements.

Game Logic

This component handles the logic of the Pool Game itself. This includes the game loop and the inner state of the game (such as keeping tracking of the Players, their assigned ball type, and their potted balls). The component makes use of the already instantiated components of the Rendering & Environment to keep track of the current state of the Game, and progress the Game further.

The component contains a single package which handles all of the aforementioned functionality.

Analysis

Pros & Cons

+ The chosen layered architecture separates the concerns of the game into 3 parts, each of which handles its own tasks, making smaller subparts of the code far easier to understand.
+ High level of testability of the architecture due to ease of stubbing out components in another layer or package.
+ High level of maintainability due to clear separation of concerns. If a change needs to be made, we can trace the change to the specific layer according to the layer's concern, then the package, and then the class.
+ Since the Game Rendering & Environment, the component that handles presentation, does not directly depend on the Storage & Logic layers, the two layers can be swapped out with ease (e.g. a cloud storage solution could be integrated instead of the current one, if desired)
- The overall view of the system might be quite hard to understand due to separation into layers and smaller packages of relatively small granularity.
- Since the game was made as a monolithic application, it is not scalable if the game were to be extended to allowing higher scale online play, because all of the functionality is handled by 3 layers on a single machine. For instance, the Game Logic would have to be handled on a single machine for all games in the current setup.
- The middle layer (Rendering & Environment) is quite a large component in the sense that it handles quite a few things simultaneously, which makes the layer rather complicated to understand.

Motivation & Comparison with other patterns

- Ease of dividing work in the team: Certain team members could be working on one layer with a very small number of conflicts between the other layers, making code changes highly productive.
  - For this reason, we opted for the layered architecture instead of the **Main program and Subroutines** pattern, which would have made modifications far harder due to the program mainly being in one place.
- Primary focus on local multiplayer, local database application: Our intention was to have the game run locally, offline, with no connection to external services required. Due to this, we chose the monolithic layered architecture over the alternatives.
  - **Publish-subscribe pattern:** Due to the primary focus on local multiplayer, we did not opt to refactor our system into a publish-subscribe pattern, which is tailored to support multiple subscribers which we did not require.
  - **Client-server:** While the pattern would make sense if we were to extend to online play, we did not opt for this pattern due to the high focus on scalability and external interaction with the server.

- Dividing the program into **layers** could also aid us in dividing the program into **tiers** later on if we ever aim to increase scalability. For instance, the database layer could be put on it's own tier. This makes our layered architecture extendable into a **multi-tier** architecture, which would allow us to move from local multiplayer only if we decided to do so with decreased complexity.

- Low risk: Due to the lack of knowledge on system design at the beginning of the course, and high familiarity with a layered architecture, we chose it to have a far lower risk of critical software issues and implementation bugs during the development process, which in the end worked out really well.
  - Since we were confident and highly familiar with the current type of architecture, we did not opt for the **Ports and Adapters** or **Service-Oriented** patterns, which we felt like had a large learning curve which would have increased development time significantly
- Complexity: Due to the clear nature of the layers and package separation of the game, we decided to opt for the current architecture over other highly feasible but more complex architectures
  - This mainly applies to the **event-driven** architecture, which would highly make sense in the context of our game, but for the scope of the project and the complexity of the system (as well as unfamiliarity) was decided to not have been worth the risk.