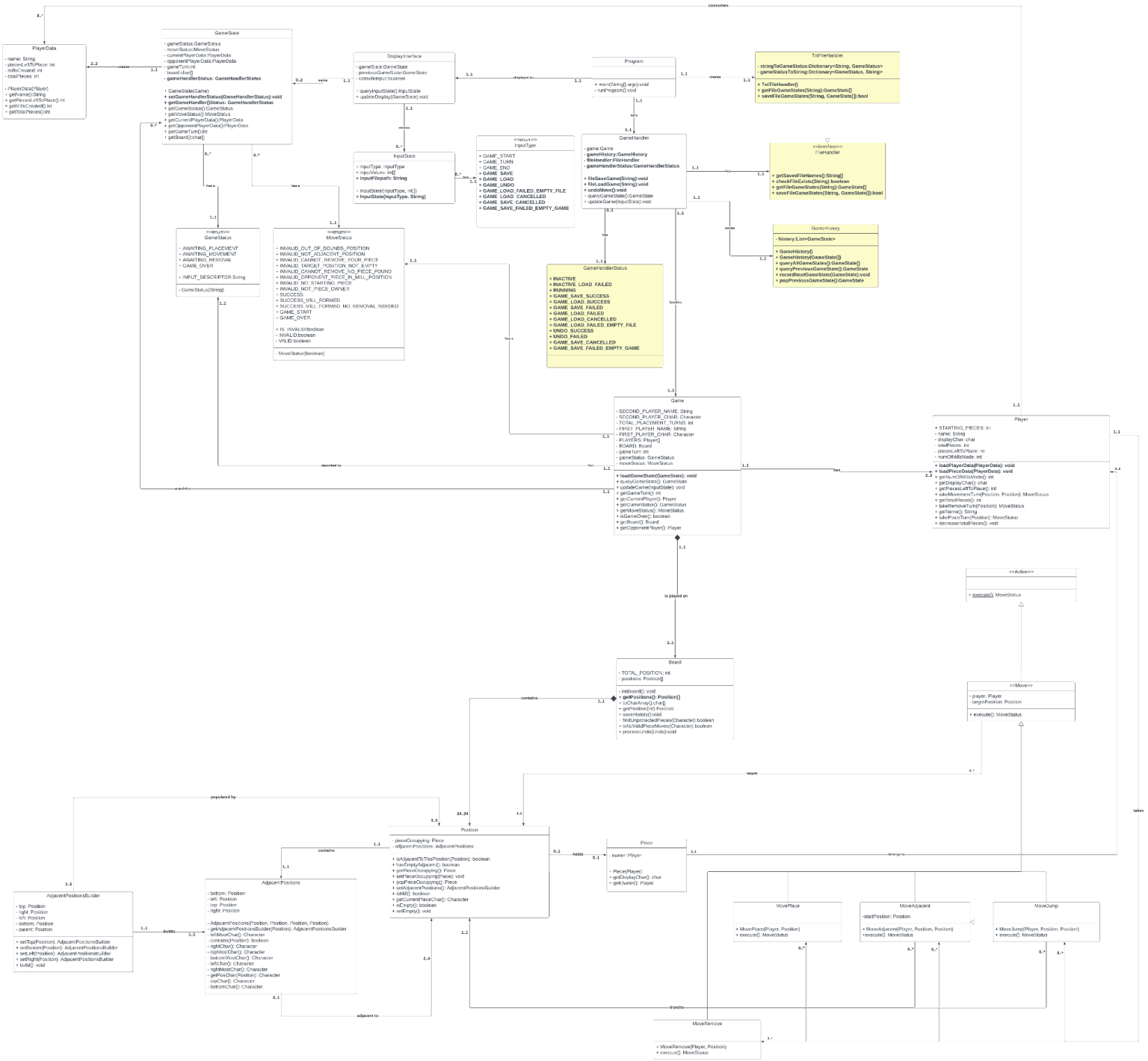
FIT 3077 - 9MM Sprint 4 - Group 42

Part A: Advanced Functionality User Stories.

- As a game, I would want to restore a previous gamestate so that I can be replayed from that point.
- As a player, I want to undo previous move/s that have been made before, so that I can replay some of them.
- As a player, I want to be able to save and load previous game states so that I can continue playing from that game state.
- As a player, I want to get feedback when my attempt to save or load is failed so that I can retry the attempt.
- As a player, I want to get feedback when my attempt to undo a turn is failed so that I know I am not able to undo this turn.
- As the display interface, I want to be able to query and get user input for saving and loading game state for the current game so that I can send it back to the rest of the program.
- As the display interface, I want to be able to query and get user input for undoing previous turns for the current game so that I can send it back to the rest of the program.
- As a game handler I want to be able to process player input for saving and loading my game so that I am able to meet the input request successfully.
- As a game handler I want to be able to process player input for undoing the previous move made in my game so that I am able to meet the input request successfully.
- As a game handler I want to be able to handle invalid file saving and loading cases so that I don't crash unexpectedly.
- As a game handler I want to be able to handle an attempt to undo a turn that doesn't exist so that I don't crash unexpectedly.
- As a file handler I want to be able to save a game state to a file so that the game state becomes persistent in memory and may be loaded at a later time.
- As a file handler I want to be able to load a game state from a file so that game handlers and players can continue games from that game state onward.
- As a file handler I want to be able to tell the game handler when their attempt to save or load a file is invalid so that they can handle the case appropriately.
- As the txt file handler I want to be able to specifically process txt files when I am asked to load or save a file so that txt files are able to be processed by the program.

Part B: Revised Architecture



Part C: Design Rationales

Advanced Architecture Design Rationale

The architecture of our game has been meticulously designed to incorporate advanced features seamlessly while minimising the need for extensive modifications to the existing codebase. By adopting a modular approach, we have divided the game into key components: the Game Engine, Display, and other in-game elements. This modular design provides a strong foundation for accommodating future code mutations and the addition of new features.

One of the key design choices that greatly influences the implementation of advanced requirements is the utilisation of game state. Rather than running the game in a linear and continuous cycle, our architecture employs the concept of game state. This approach involves injecting the previous state of the game into the game engine and retrieving the new game state after each turn. By implementing this mechanism, we establish the groundwork for implementing features such as an undo function. As the game data is saved at each turn, the undo functionality can easily access and restore previous states from memory. To facilitate this, we have employed the Memento design pattern, which ensures that maintaining a history and modifying the undo function is straightforward.

The GameHandler serves as a proxy that facilitates all game updates. This structure allows the GameHandler to save states as they pass through it without directly affecting the game itself. It ensures that the game state remains completely separate from the game, preventing potential artefacts from file management or other operations.

Furthermore, our architecture allows for external handling of actions that do not exist within the game but can still impact the game. For example, loading and saving the game should ideally be transparent to the game itself. If the game were aware of whether it was loaded from a file or played normally, there could be a risk of introducing artefacts during the saving and loading processes. By segregating these operations, we guarantee error-free loading and saving of the game, with any potential errors confined to the file handlers themselves.

To maintain this separation of concerns, the file handlers are directed from within the GameHandler. While it may be argued that a separate GameLoader/GameSaver class could act as an additional layer of proxy, the current proxy setup is already fully functional and allows for further expansion of saving and loading functionality through the implementation of concrete file handlers. This approach is achieved using an abstract interface, ensuring that the agreement between the GameHandler and the file handlers remains intact and does not require any additional modifications. Introducing another layer of separation would render any changes to the current GameHandler proxy redundant and unnecessary.

In summary, our thoughtfully designed architecture incorporates advanced features from the beginning, provides a modular structure for easy addition of new functionalities, utilises

game state and the Memento design pattern for efficient management of undo functionality, employs the GameHandler as a proxy for handling game updates and state-saving, separates external actions from the core game logic to prevent artefacts, and directs file handling operations through the GameHandler to ensure a streamlined and scalable approach.

Revised Architecture Rationale

In our updated architecture, we implemented several changes to accommodate the advanced requirements necessary for the final prototype. The key modification involved broadening our design to handle not just the internal game states, but also the saving and reloading of game states from text files.

In the initial architecture, we concentrated primarily on internal game state management, particularly undoing the game state. This focus was apparent from the addition of the 'History' and 'Undo' classes, which temporarily stored data while the game was in progress. However, this design did not account for the functionality allowing the game client to save an active game's state and reload any previously saved games. To address this, we introduced the 'Interface Class FileHandler', equipped with 'getFileGameStates()' and 'saveFileGameStates()' methods to facilitate the saving and reloading of game states. Furthermore, we created a subclass, 'TxtFileHandler', to specifically handle the storage and retrieval of game state data in text file formats. This inclusion offers flexibility for potential future requirements necessitating different file formats.

A few changes were also made to facilitate the loading of a game state within a game to meet these requirements and have also been highlighted in bold on the class diagram alongside the other changes. These involve a method to load a gamestate data object back into a game object, and functionality for loading player data into a player, and board data into a board. Despite these additions, they are only that, additions and do not modify preexisting code from previous sprints which really shows the power of how we have set of the rest of the codebase, particularly in that the gory details of IO do not require any modification of the interal game logic layer. This can be seen in the class diagram where no pre-existing functionality is highlighted in bold as none of it was changed to ensure the advanced functionality works as required.

We retained the 'GameHistory' class as the primary recorder and saver of game data, but we removed the 'Undo' class as its functionality was superseded by the file handling design. We also introduced the 'GameHandlerStatus' Enum class to mirror the existing 'GameStatus' Enum class. This new class aids in maintaining code consistency and handling error messages.Through these revisions, we ensured that our architecture fulfils all the advanced requirements while providing a solid foundation for future extensions. The improved design is not only robust and flexible but also more capable of addressing the complexities of game state management and file handling.

As a final note for this section, it was given in feedback that we should ideally separate out project into packages this was considered but decided against and should have been

included in the previous sprint's rationale. As, while this might make sense from a modularity standpoint, it goes against what java packages are meant to be used for, that being facilitating modular reuse in other repositories, libraries and frameworks. If this repository was going to be used by other repositories who would only require parts of this project then packages should be used. However since this project exists in its entirety and should only run in its entirety to add packages would be anti-intuitive to someone using it who may think that parts of the project are designed standalone or to be used without the rest of the project.

Advanced Requirements Finalisation Rationale

Our advanced feature was finalised from the outset, in sprint one. Although we initially considered implementing the Hints/Tutorial advanced requirements, we unanimously decided to commit to the Undo/Reload game state advanced requirements, as evidenced by the class diagram in our first submission.From the beginning, we intentionally factored in the anticipation of the undo functionality by using the game state to store data on a turn-by-turn basis. This foresight made it straightforward for us to implement the feature, particularly through the use of the Memento design pattern. All the changes in this sprint are addition to the code base and not major mutations that have resulted in complete rewrite of the code.

One key change however was moving the memento handling to game handler, the motivation for this stems from realising that actions on the game class should be handled outside of the game class, as the game class already has the responsibility of orchestrating in game functionality. It is also driven by the separation of technical implementation from game logic, wherein the handling of the file system shouldn't happen in the game itself. Moving the advanced functionality to the game handler resolves these issues for the most part and overall results in a more compartmentalised final codebase.

Throughout the project, we adhered to good design practices and patterns, including Singleton and Facade, and continuously worked to reduce dependencies and various code smells where possible. This adherence to best practices kept our code clean and flexible, making it easier to adapt and mutate when necessary. Consequently, we didn't find it difficult to implement the advanced feature; in fact, it was relatively easy due to our thoughtful planning and commitment to good design principles from the early stages of our project.