Sprint 4 Rationale

FIT 3077 Group 29
Tycl's inc. ©
Terry Truong 32466870
Cynthian Thai 31477119
Leon Ma 31449840
Yuhan Zhou 31468748

Revised User Stories

- As a user, I want to be able to save my game when I am quitting so that I can return to the game later.
- As a player, I want to be able to undo so that I can remove a mistake that has been made.
- As a player, I want to be able to repeatedly undo moves to remove an earlier mistake that has been made.
- As a player, I want to be able to play a tutorial to learn how to play the game.
- As a player, I want to be able to use hints for valid moves so that I can see where I can make a valid move
- As a UndoMove Action, I want to be able to access the most recent moves from a given player in order to reverse this move
- As a Player's Class, I want to be able to keep a history of all the moves that have been made so that I can see which moves I have made previously
- As a GameExporter, I want to be able to know which file I am writing to and from in order to access the correct file.
- As Game, I want to be able to know game board information in order to restore the previous board
- As Game, I want to be able to know previous player information in order to restore the player's information
- As UndoAction, I want to know which undo move is possible in the current state in order to execute the appropriate undo move.
- As Action Class, I want to know what action the player performed in order to execute the appropriate action.
- As GameExporter, I want to access player information in order to write this to the save file in order to recreate this information when the player returns.
- As player, I should not be able to load a game with no previous save slot in order to know that the game was not saved previously
- As HintAction, I want to know which hint can be provided in order to provide the correct hint to the player.

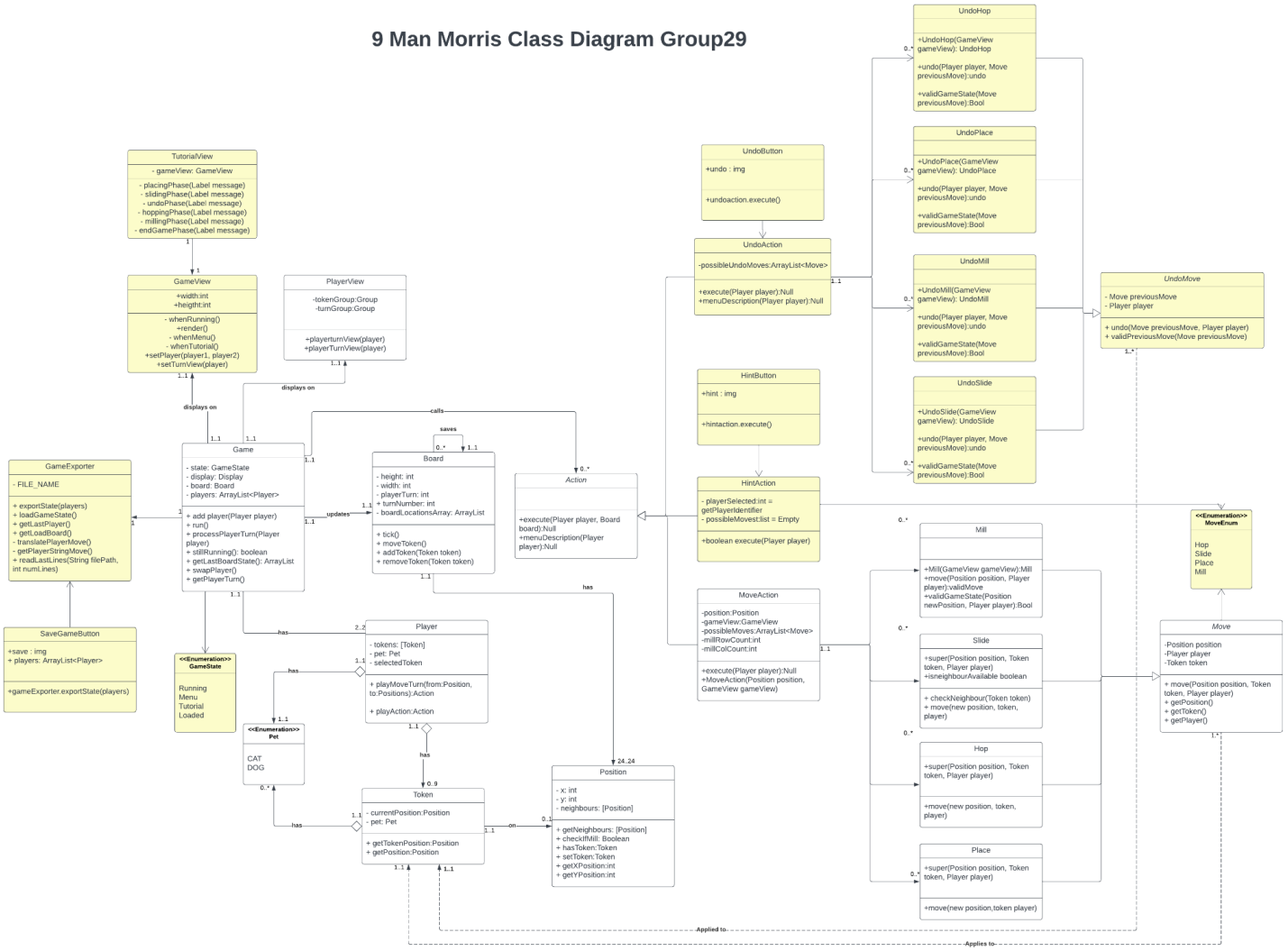
Building and Executing Executable

1. Download JavaFx JDK appropriate to machine architecture
2. Run the executable file (.jar file) within the terminal through this command:

java --module-path <path of JavaFX SDK directory>\lib--add-modules javafx.controls,javafx.fxml -jar <path of JAR file>

Architecture and Design Rationales

Revised Architecture Class Diagram



Design Rationales

Architecture design for the advanced requirements

We designed the tutorial mode as a separate option from the main gameplay as we believe the tutorial should not be incorporated into the start of every game session. This is because, for those who are already well acquainted with the rules of the game, they may find it a hindrance to constantly be informed of the rules. Due to this reasoning, we gave the option to play the tutorial in the main menu along with an option to start a game.

Furthermore, we have implemented the hint button, which triggers the execution of the hint action class. The hint action class, an extension of the action class, utilises its execute method to offer a "clue" to the player upon activation. The hint button functions effectively, displaying to the player the available moves specific to the current stage of the game. This feature ensures that players are informed of the appropriate moves they can make at different stages, enhancing their gameplay experience.

The implementation of UndoAction followed a similar architecture to MoveAction, where different undo move classes: UndoPlace, UndoSlide, UndoMill and Undo hop extended the abstract class of UndoMove. This allowed the logic of valid undo moves to be passed to the move class instead of the Token or the Game class. Due to the architecture of MoveAction being carefully revised throughout sprints 2-3, it has allowed UndoAction to be integrated easily into the existing architecture with only needing to change the different elements whether it be players, positions, moves that need to be passed instead, other aspects remained the same.

Explanation for Revised Architecture

The main architecture was not changed much apart from adding setters, getters and some helper methods for easier calls to pre-existing classes. Our advanced functionality required us to add new classes with their respective functionalities and code.

A few button classes have been created to allow the user to prompt certain actions such as UndoAction and HintAction . New methods have been added to the GameView as there are multiple game modes including Tutorial mode and a game Menu. A GameState Enum was added so that the gameView can know which view to display to the user given the state whether this be the menu, game board or tutorial. This also includes a TutorialView class being created that contains methods for running the tutorial. It handles the changes to the game and game view when the player learns the game rules. Similarly, saving the game state to a text file is handled by the GameExporter class which also allows the current game state to be saved and the previous game to be loaded.

Furthermore, we have discovered during this sprint that it is more suitable to pass our previously labelled game states (Slide, Place, Hop, Mill) as an Enum rather than as strings, which was our previous approach. By adopting this new approach, we not only improve the appropriateness and efficiency of our method but also enhance the overall clarity and

readability of the codebase. The utilisation of an Enum enables us to achieve consistent and standardised representation of the game states across our system. This, in turn, fosters improved comprehension and easier maintenance of the code, promoting better overall understanding and long-term manageability.

Finalising advanced features, Ease/Difficulties of implementation

In Sprint One, the team decided on the advanced functionality of undoing moves and allowing save games and having hints and a tutorial mode. We reflected this in our class diagram since sprint two and have stuck with the implementation through Sprint Four. Throughout sprints 1-3 of the project, the existing architectural and design practices were revised numerous times for prospective enhancements. UndoAction, which behaved similarly to MoveAction, was easily implemented as a result of these optimisations. UndoActions can easily be created and executed within the game by adopting a similar design pattern, passing on the logic of valid undo actions to the responsibility of the undo move class itself. It also demonstrated the architecture's adherence to the Open and Closed Principles.

For the advanced features, we followed the Single Responsibility principle in SOLID principles which allowed our advanced functionality to not require the team to rewrite the majority of our pre-existing code. The advanced features however did introduce some challenges as the pre-existing code was not optimised in previous sprints to be prepared for the advanced functionality. This resulted in the advanced functionality's planning being somewhat difficult as we had to plan out. For example, the main game was not prepared to have a tutorial mode that would have an example board set up to allow the user to try different phases of the game without actually getting to those stages naturally. We resolved this by introducing a wipe to the board game state after the tutorial is run so that it would not affect the currently running game state when starting a new game. We also added logic to randomly generate a board with specific parameters so that the tutorial can demonstrate each respective state of the game. Ideally, the user would be able to have multiple Game instances running if they wanted, however.

Other challenges faced also involved being unfamiliar with file management, reading and writing to files within Java was unexpectedly more complex in comparison to Python where intermediate steps such as specifying the number of bytes to write and setting up file readers were omitted. Furthermore, having not worked with Javafx previously has also made creating an executable file difficult as fx: deploy has terminated its support for JDK version 11+.