Sprint 4 Deliverables

Design Rationales

Advanced Requirement (B - Undo Moves and Save Games)

Explain why you have designed the architecture for the advanced requirement(s) the way you have.

Our team has decided to implement advanced requirement B, undoing moves and saving games, for our game in a web browser environment. After considering the security restrictions that prevent direct file system access from JavaScript in the browser, we explored two feasible options: 1) server-side file handling, and 2) local storage and download files. After a team discussion, we have chosen server-side file handling as the preferred option, with our discussion and evaluation summarised below.

1. Server-side File Handling

   With server-side file handling, we have set up a server using Node.js and Express. When the player wants to save the game, the game state is serialised and sends a request to the node server. The server then writes the data to a text file named "data.txt".

   When the player opens the menu page, it will load all the previously saved games to display in the game list by sending a request to the server. The server reads the *data.txt* file and sends all the previously saved games and their game states back to the menu page. The games are then loaded into the game list for the player to select and resume playing.

2. Local Storage and Download Files

   In this approach, we would store the game state as JSON in the local storage of the user's browser. To facilitate saving and loading, we would provide an option for the user to download the game state as a text file. This means we would not be using a text file directly for the save/load functionality.

In order to implement the saving and loading functionalities in a web browser environment, our team decided to use server-side file handling with Node.js. The reasons why server-side file handling is the preferred option include its abilities to store data, accommodate future file format changes and overcome the security restrictions of the web browser environment.

1. Storing Ability

   In terms of storing game data, using a text file for saving and loading the game is more efficient than using local storage. Server-side file handling allows our 9 Pets' Morris game website to simply send a request to the server to save the current game state, and the server can handle the file writing. Similarly, when we want to reload previously saved games, the application can request the *data.txt* file from the server and load the games.

   On the other hand, local storage is limited to storing data on the user's device within the browser. While it can be used to save small amounts of data, it is not well-suited for saving the entire complex game state. Using local storage might be challenging and inefficient.

2. Accommodating Future File Format Changes

   Considering that different file formats may be required in the future, server-side handling logic resides on the server, therefore, it can be updated or extended to support different file formats without directly impacting the game-side implementation.

   Local storage is not designed to handle different file formats. If there is a need for storing game data in different file formats in the future, local storage would not provide the necessary flexibility and extensibility.

3. Overcoming Security Restrictions

   The security restrictions imposed by the web browser are due to the lack of permission for direct file system access from JavaScript running on a web page. With server-side file handling, the server can handle the file operations, ensuring secure and controlled access to the game data file.

   While local storage provides a means to store and retrieve data on the user's device, it does not overcome the security limitations that prevent direct file system access.

In conclusion, by choosing server-side file handling, we address the security restrictions of the web browser environment, ensure efficient storage and retrieval of all games, and allow for future compatibility and extensibility.

Explain when your advanced feature was finalised and how easy/difficult it was to implement.

The advanced requirement (B - undo moves and save games) we implemented is the same as we decided on in Sprint 1.

Overall, implementing the advanced features was relatively smooth, but there were some unexpected challenges along the way. This can be mainly attributed to our detailed planning and preparation in Sprint 2, where we already brainstormed how we would implement the advanced feature.

From our thorough brainstorming, we had a solid understanding of how to implement the advanced features of undoing, saving and loading existing games. In fact, we already had empty methods (*undo*, *save* and *loadGame*) for such features, just waiting to be filled out in this sprint.

Why our chosen design in earlier Sprints helps facilitate the implementation of advanced features is evident through our decided structures for the Game, Team and Position classes and how to save the data/states of a game.

For instance, since we already knew from Sprint 1 that we would need to implement the advanced feature of undoing moves, we have *boardHistory* as one of Game's attributes, which stores the board that reflects the game state of each round in the game. So instead of simply updating the same board for each round, we duplicate and push the game board for each round into the board history. This allows us to undo the last move by simply popping the last game board from the board history and updating the display, which will then revert the game to its previous state.

Another piece of evidence is our decision of having Team and Position as their own classes. This allows us to easily convert Team and Position objects to JSON, and vice versa, when we are implementing the advanced feature of saving and loading games.

The biggest challenge we faced when implementing the advanced requirement was how we would handle the game with the advanced features under the web environment, especially for saving the game states in a text file. Originally, we planned to directly write the game states to the *data.txt* file. However, this would not work due to the lack of permission for direct file system access from JavaScript running on a web page. To solve this problem, we used the NodeJS architecture which would prove to be a challenge as none of us had much experience with it. We had to create another file to facilitate this communication and change up some methods in other classes to accommodate this change.

Revised Architecture

The architecture has mainly remained the same as there was no addition or change of classes or relationships; only a few methods and attributes have been added to certain classes to facilitate the implementation of our advanced requirement (B - undo moves and save games).

The reasons for the addition of methods and attributes in each class are detailed below:

Application

- showingMenu

  This attribute was added to indicate whether the application should be showing the menu page or the game page. Without this attribute, the application will not know whether to show the game list for the menu page or the game board for the game page.

- loadApplication()

  This method was added to facilitate the loading of the application. It is used to either load the menu page or the game page, based on the value of *showingMenu*.

- createNewGame(currentGameIndex)

  When implementing the advanced requirement of saving and starting new games, this method was originally a part of the *loadApplication* method. To improve code modularity, we have separated the logic for creating a new game from loadApplication.

- downloadGameData()

  This method was added to implement the advanced requirement of saving, so that the players can have the option to download the data of all their games into a simple text file.

Game

- constructor(boardHistory, gameIndex, currentBoard, name)

  The change to the Game constructor is the addition of *gameIndex* and *name* parameters. The reasons for adding these two attributes are listed below.

- name

After implementing the advanced requirement of saving, the users will now be able to save their games. Adding the *name* attribute allows the players to add a name for their game when saving it, so that the game can be easily identified by the players.

- gameIndex

  As the advanced requirement of saving allows more than one game to be played and saved to the application, this attribute is used as a unique identifier for each game by the application.

- getBoardHistory()

  This getter was added to allow the application to save the board history when saving a game. It is also needed for checking *boardhistory* and disabling the undo button if no previous moves are available.

- getName()

  This getter was added to allow the application and display to retrieve the name of each game, so that the players can identify the different games in the game list.

- exit()

  This method was added to allow the players to exit the current game and go back to the menu page, as per the advanced requirement.

- save()

  This method provides the application a way of saving the game states of the current game before exiting, so that the game can be reloaded and resumed by the players later.

- saveToFile(gameName, gameIndex)

  This method was added to save the game state of the current game into a simple text file, which is a part of the advanced requirement. It is called inside the *save* method.

Display

- setUpButtonListeners(game)

  This method was added for setting up listeners for the new exit, save and undo buttons, which were all required and added to implement the advanced requirement.

Board

- setUpPositions(positions)

  The change to this method was the addition of the *positions* parameter. Before implementing the advanced requirement of loading previous games, we only needed to set up an empty board when starting a game, hence no parameters were required. Since the players can now reload any previous games, which will require a board that reflects the game state when they last left the game to be set up when resuming the game. The *positions* parameter is used to set up non-empty game boards.

- joinPositions()

  This method was originally a part of the *setUpPositions* method. To increase code modularity, the logic for joining the positions with their neighbours was separated into this method.

- getTeams()

  This getter was added to facilitate the saving of a game, as the application needs to know all the teams that were involved in the game.

- getCurrentPlayer()

  This getter was added as we realised the player of the current round in a game is often required.

- toJSON()

  This method is required to parse all the data of a Board object into JSON to facilitate the saving of a game.

Position

- toJSON()

  This method is required to parse all data of a Position object into JSON to facilitate the saving of a game.

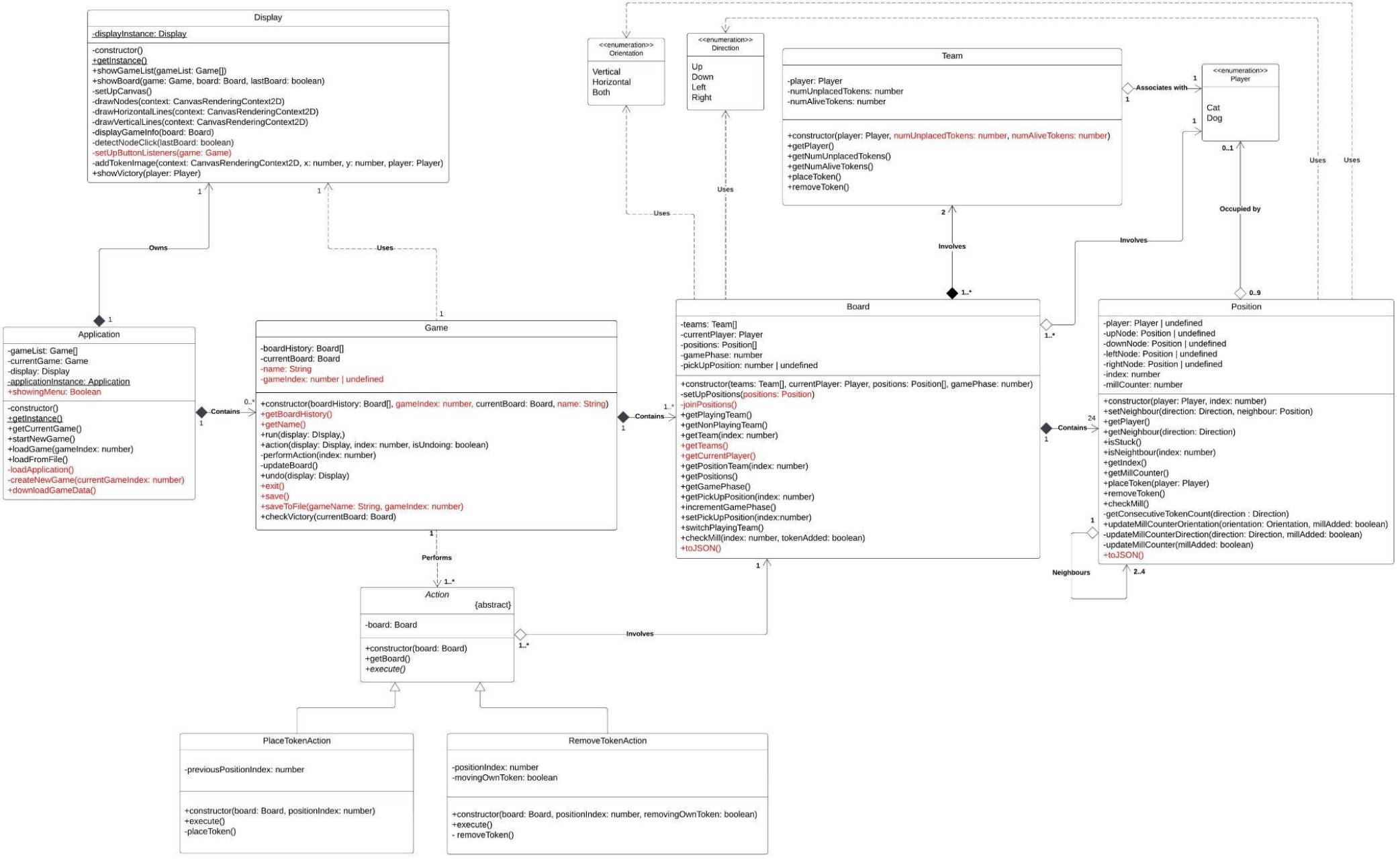
Team

- constructor(player, numUnplacedTokens, numAliveTokens)

  Before the implementation of reloading previous games, all teams were set up with 9 unplaced and alive tokens by default. By adding the two parameters, *numUnplacedTokens* and *numAliveTokens*, to the Team constructor, it allows teams to be set up with a specific number of unplaced and alive tokens when the players reload and resume a game, to reflect the game state when the players left it last.

Revised Architecture

All changes to the software architecture are written in red.



Revised User Stories

Four new user stories for the advanced requirement have been added, which are in yellow. (All other grey user stories remained unchanged from Sprint 1.)