FIT3077 - Software engineering: Architecture and design
Sprint 1 - Rationale

CL_Monday6pm_Team37
Team: HTML is an OOP Language

Index

Team Information

Team Name and Team Photo

HTML is an OOP Language

Cyrus, Michelson, Sean (left to right).

Team Membership

Member Name	Email	Technical/Professional Strengths	Fun Fact
Michelson Fu	mfuu0004@student.monash.edu	Python, questionable team leader /S	Own a Cat.
Sean Wong	swon0091@student.monash.edu	Proficient in Java, Python, SQL, good team player	I play Vanguard and Destiny 2
Yin (Cyrus) Lo	yloo0010@student.monash.edu	Java, Python, C++, good team player	Fluent in Japanese. (Not first language)

Team Schedule

Regular Meeting Schedule and Progress

- **March**

15/03/2023 - Group Formation
17/03/2023 - Group Initial meetup
21/03/2023 - Meetup for Sprint one brief released
28/03/2023 - Meetup for updating sprint status
30/03/2023 - Meetup for rounding up the work
02/03/2023 - Meetup for polishing the work

- **April**

03/04/2023 - Due Date of Sprint One

Note: Our group members' availability vary from week to week, but we aim for 2-4 hours per week of meeting and work time in addition to the workshop contact hours. We usually have 2-3 hour blocks scheduled on either Tuesday or Thursday evening.

Workload Distribution
We aim to distribute workload evenly, team members will contribute to all parts of the project. All members were present in all meetings and working sessions and all contributed to all parts of the project. For details, please refer to the wiki on GitLab.

Technology Stack and Justification

Considered Programming Languages	**Pros**	**Cons**
Java	● Java is a fully fledged Object Oriented Language ● Platform-independent ● All members are familiar with it from prior units ● LWJGL	● Requires more complicated external frameworks to achieve similar level of 'plug and play' that can be achieved with Python
Python	● Everyone is well familiar with the language. ● Python has a module named 'pygame' which could be used to implement the UI of the game	● Not a fully object oriented language ● Slow speed ● High memory usage
C++	● Object Oriented Language ● Low-Level Manipulation ● Precise memory management	● Pointers ● Lack of Garbage Collector ● Not everyone has experience with C++
C#	● Object Oriented Focus ● Cross platform of .NET ● Garbage Collector ● Easy to learn if you know Java	● Performance ● Heavily rely on .NET
TypeScript	● Improved JavaScript ● Type System	● Complicated ● Team member have no experience with it

Final Choice of technologies used
We were deciding between Java and Python as our choice of programming language as those were the languages that our team members are most confident with. However, considering the game library as well as our team having limited understanding of Java's LWJGL, we ultimately decided on using Python as it features easy UI development with the Pygame library, which is much easier to use compared to Java.

The alternatives were discarded due to team members having very little experience as well as confidence using them. This was because we were comfortable with and confident that we could use either Java or Python to implement the required deliverable based on the specifications given to us.

Additional Requirements for Final Prototype:
A. Tutorial mode and "hints" toggle of legal moves
Our approach: Create different board states representing each phase of the game and provide single possible moves so that players can observe all possible actions at each stage of the game. The hint toggle will consist of circling the currently selected piece and valid moves for that piece in green.

User Stories

We will go through each user story that we think of, and evaluate them using the INVEST criteria.

1. As a player, I want to be able to view the tutorial so that I can learn the rules and understand how to interact with the board.

   Independent: The tutorial is standalone and not interfering with the game code.
   Negotiable: The content of it can be scaled depending on the needs.
   Valuable: The user can learn from going through the tutorials.
   Estimable: The game rules are not infinite
   Small: Small enough.
   Testable: Functional test of media playback

2. As a player, I want to have a start game button so that I can start playing a game (against another player)

   Independent: The event listener of the button can be mapped to do other stuff.
   Negotiable: The action that the button does can be changed.
   Valuable: The user can be able to play a game of Nine men's Morris.
   Estimable: Yes
   Small: Just a button that links to the game.
   Testable: Functional test of can it initiate a game

3. As a player, I want to place pieces, so that I can set up the game (game phase).

   Independent: Once the board is initiated, this action is independent to others.
   Negotiable: It is one of the actions required to play.
   Valuable: The user can do one of the moves.
   Estimable: Yes
   Small: One of the actions when playing.
   Testable: Functional test of can it place the piece successfully.

4. As a player, I want to create a mill, so that I can remove an opponent's piece and get closer to winning.

   Independent: The code only needs to recognise when a mill has been created.
   Negotiable: The way that the recognition of mills differs.
   Valuable: The user can do one of the moves, and remove an opponent piece.
   Estimable: Yes
   Small: One of the actions when playing.
   Testable: Functional test of can it recognize a mill correctly.

5. As a player, I want to know who's turn it is, so that I know when it is my turn to play.

   Independent: The code should already be tracking who's turn it is, we just display it.
   Negotiable: The way that indicates whose turn differs.
   Valuable: The user could know whose turn it is and not be confused.
   Estimable: Yes
   Small: Just trigger an element to be visible or not.
   Testable: Functional test if it display the turns correctly.

6. As a player, I want to know the result of the game, so that I can know the game has ended and who has won.

   Independent: A result screen will not interfere with other components of the game.
   Negotiable: The way of displaying the result can be considered.
   Valuable: The user could know that they have won the game.
   Estimable: Yes
   Small: Just a screen to display and exit.
   Testable: Functional test if it displays the correct winner.

7. As a player, I want to be able to select my pieces, so that I can move them.

   Independent: Once the piece is on the board, we only need to recognise the click.
   Negotiable: One of the main game actions.
   Valuable: The user could play strategically when moving the piece.
   Estimable: Yes
   Small: Just recognise which piece the user selected.
   Testable: Functional test of can it select and give visual feedback.

8. As a player, I want the piece I select (that belongs to me) to glow green or red so that I know whether it has valid moves.

   Independent: Only need to implement the glow for individual pieces.
   Negotiable: The glow differs.
   Valuable: The user could know if they can move this piece or not.
   Estimable: Yes
   Small: Only need to check if that piece can move or not.
   Testable: Functional test of correct visual feedback.

9. As a player, I want a hints function that shows me possible valid moves, so I can consider ALL possible valid moves.

   Independent: The algorithm needs to check where it could be placed.
   Negotiable: The way of displaying this information varies.
   Valuable: The user can be supported by the hint.
   Estimable: Yes
   Small: Only need to check and display valid positions.
   Testable: Functional test of giving visual feedback of the correct position.

10. As a player, I want to have an exit button so that I can exit the game.

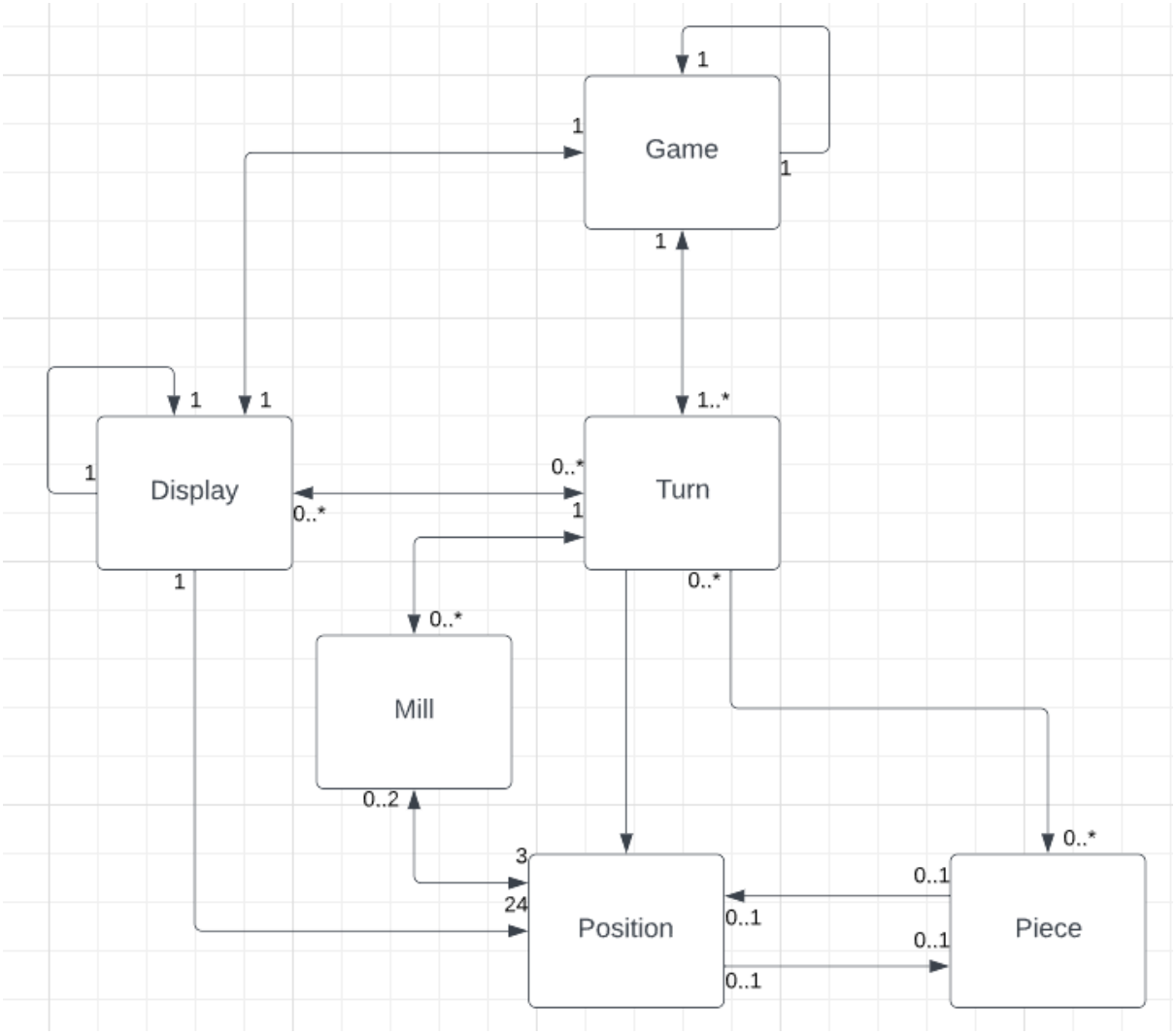
    Independent: Just an exit button to the menu screen.
    Negotiable: How the exit button is implemented differs.
    Valuable: The user will not be stuck at the screen.
    Estimable: Yes
    Small: Just a button.
    Testable: Functional test of can this button return to the main menu.

Basic Architecture

Domain Model

We did our draft of the domain model using Lucidchart. Here is the link to it.
https://lucid.app/lucidchart/0abd0599-3e18-4457-be1e-996aeab66dd3/edit?shared=true&page=0_0#

See next page for the diagram and explanation.



Explanation:
Game is the class responsible for the game logic and coordinating the other classes so that the game can be played. It does this by switching turns between players and updating the game state and display depending on the return of Turn (which tells Game what has happened on the turn), telling the other player that it is now their turn.

Display is responsible for telling what the player has requested to do, and updating the UI by reading what changed on the board in Position. Display may also tell Game to start a new instance of itself (on playing a new game).

Turn is responsible for telling the display to show which players' turn it is as well as what moves they are allowed to make or if they have to remove an opponent's piece (because they formed a mill). Turn is also responsible for moving a Piece (or removing it) and updating Position depending on player input through Display.

Position is responsible for storing whether a position is empty or contains a piece and what colour the piece is (which player it belongs to) as well as its neighbours..

Piece is responsible for knowing what position it occupies.

Mill is responsible for checking if the last moved position (piece) affected any existing mills

or created a new one. It is invoked by Turn. A list of mills will be stored in game (passed through by Turn).

Associations and Cardinalities.

Game → Game:
Can check if itself already exists. Can create an instance of itself if necessary.
Game → Turn:
A game will have many turns, but a turn made in a specific game only belongs to that game that it was made in.
Turn → Game:
Tells the Game what turn has been made. (Turn belongs to 1 game).
Turn → Piece:
A turn could affect more than one piece (moving a piece to form a mill then removing an opponent's piece). A piece can also exist (on the board) without being affected by a turn.
Turn → Position: Turn checks if the most recently moved piece has formed a mill and then takes necessary action. It can also check if the positions surrounding a are occupied by other pieces by looking at Position.
Turn → Display:
There are many different display states and turns.
Position ↔ Piece:
A piece can be in a position or it can not yet be placed (beginning phase). A position can contain a piece or it can be empty.
Game ↔ Display:
A game and a display belong to each other. Game contains the game logic (code) and takes input from the display (from the user), processes it and then tells the Display to update itself by reading from Position.
Display → Display:
Can check if itself already exists. Can create an instance of itself if necessary.
Display → Position:
Display can read from Position so that it can show the correct position of pieces.
Display → Turn:
There are many different display states and turns.
Mill ↔ Turn:
More than one Mill can be affected on one turn, a piece could be moved out of a Mill to create another one. However, every mill is created or destroyed on a specific turn.
Mill ↔ Position:
Mill contains three positions (must). But a position could not be in a mill or part of up to 2 mills.

Discarded:

We thought of using a Board class, but decided that calling it or using a Position would be easier. As Position(s) would exist inside the Board class anyway. It would have been an extra entity in the Domain Model that served little purpose and convoluted things.
We could have possibly used a TurnHandler and created child classes of Turn for each of the allowed types of move (which depends on game state) such as placing, moving, sliding, removing etc. However, we decided that all of those types of turns could just be functions written inside the Turn class and called from itself.

Design Pattern

Considered design patterns that might be useful	
Factory	For turn (we make many of those hopefully).
Singleton	In Nine Men's Morris for two players, there could be one game for both players, and for each game there could be a board for them. Many of our designs will only create one instance.
Observer	We could use an observer when the token status changing at the location where could create a mill, it activates the checking to see if there is a mill or not.

Design patterns that we are not using	
Facade	Facades are very handy when we have a lot of subclass object handles. In our design of Nine Men's Morris, we have many types of objects but not many subtypes of them are introduced. We will not be implementing that many object classes.
Adapter	We are building this game from scratch so we will have no existing class. This pattern is good when there is an existing class to build up on, but since it is not the case here, this pattern is relatively less useful compared to others.
Strategy	Our project will not have different variants of algorithm and object switching simultaneously at one time. Only the tokens on the board are changing. This is not suitable for our cases.
Visitor	There will be a few actions that the players can do in Nine Men's Morris. The player only needs to place, slide, fly and remove. We are not focusing on adding additional actions to the game.

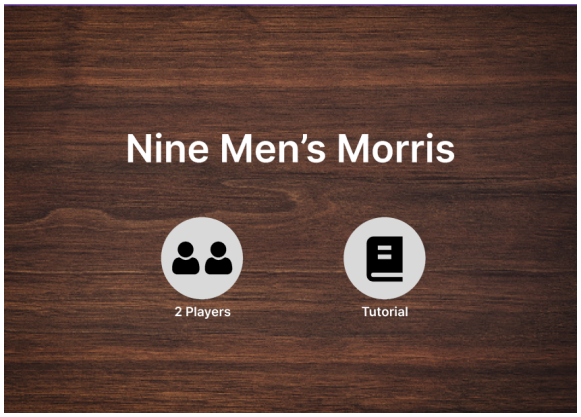
Final decision of the design pattern

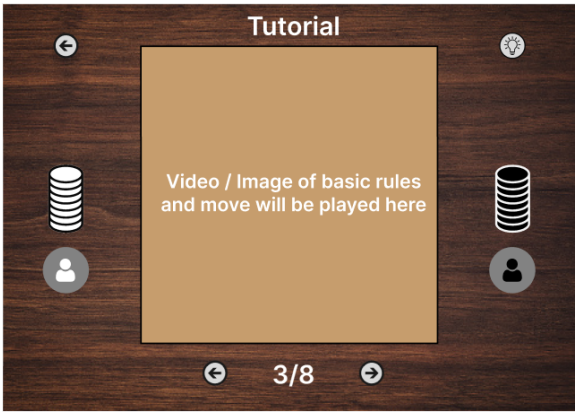
We discussed the possibilities of what design patterns are suitable for our cases of Nine Men's Morris. Singleton design pattern can be used since there would be only one game and a board with it. On the other hand, Observer can be useful when detecting mills creations, token placed, and other event listeners. That way the creation of those events are always being checked and wasting resources. Overall, we are using the Singleton and Observer designs pattern.

Basic UI Design

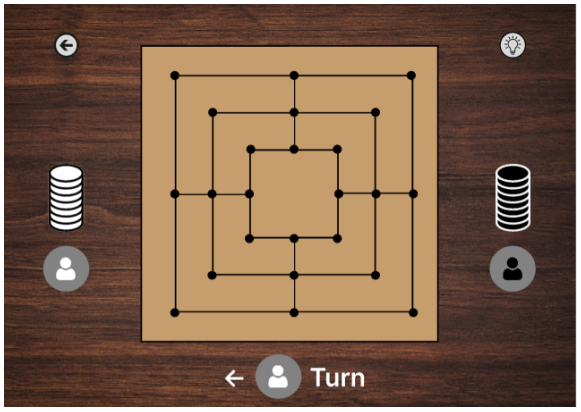
We did our UI using Figma, the images are shown below with explanation and justification. The images are sketches and may not reassemble the final product. You could look at the Figma link if you are curious.
https://www.figma.com/file/NDC6RdBleY1y3IMvP2POuM/FIT3077-Group-37?node-id=0-1&t=jVOl8MP3K9pgfFwv-0

Screens

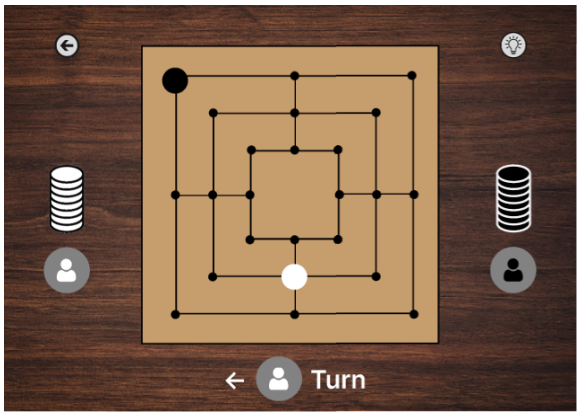
Main menu	
	The main menu contains the game's title and two buttons. The buttons lead the user to play Nine Men's Morris with another player, or go through the tutorial if the user needs any explanation for rules and moves. **Justification:** There are only two options. For simplicity, just putting them on the menu screen with a pictogram can help the user understand. For the background, wood texture was introduced to simulate playing it on a table.

Tutorial	
	This screen will show when the user clicks on the tutorial button at the main menu. To teach the user about the rules of the game, a video (or images) of gameplay will be shown to demonstrate what to do. At the bottom are the page buttons to go forward and backwards. The user could fiddle around the pages in case they wanted to go back. **Justification:** Having media explaining is a cost effective way to demonstrate without spending time coding animation with in game logic. The page button there allows us to split the tutorial into smaller, organised sections. That way the user could rewatch a section without having a hard time looking for a specific part.

The board



Initial board

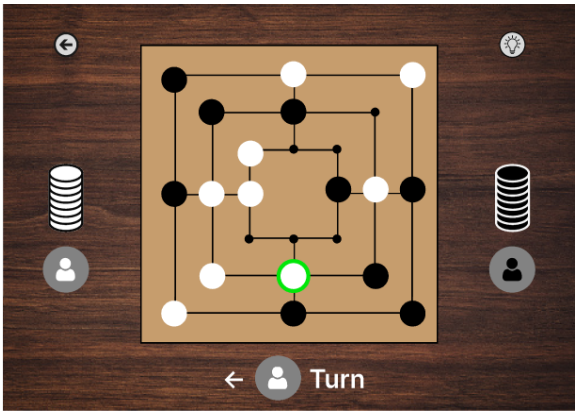


Placing pieces

The initial board will show when the game started. The board in the middle, a return to menu button on upper left, hints button on upper right. Player pieces on left and right representing the total piece that player has not placed yet. At the bottom, it marks whose current turn. When placed the pieces, it will show up on the board as so.

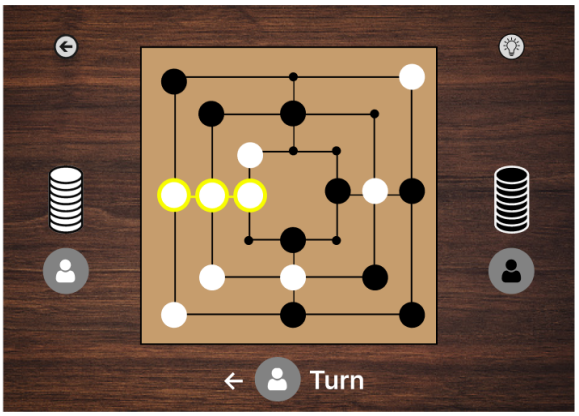
Justification:
The board is the focus of the game, putting it in the middle attracts users attention. The back to menu button is at the upper left due to most operating systems putting the back button there. This causes a natural habit of going back to the upper left corner, therefore, putting it there makes sense. The hint button is put upper right for a symmetrical look, so that it feels more balanced to the eyes. The player token piles are put left and right because both players are going to sit next to each other facing the display. Hence, left and right. For the turn indicator, supportive text like subtitles are at the bottom, this suits this situation and we put it there.

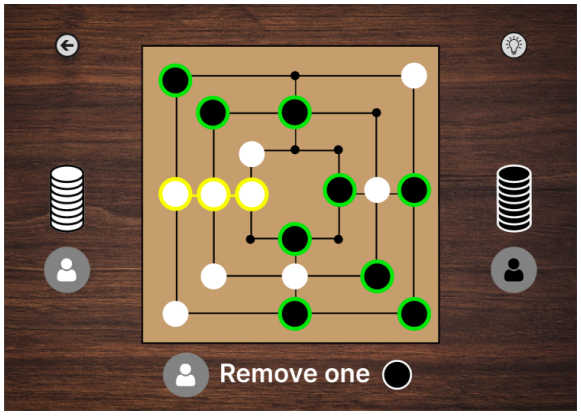
Selected a piece

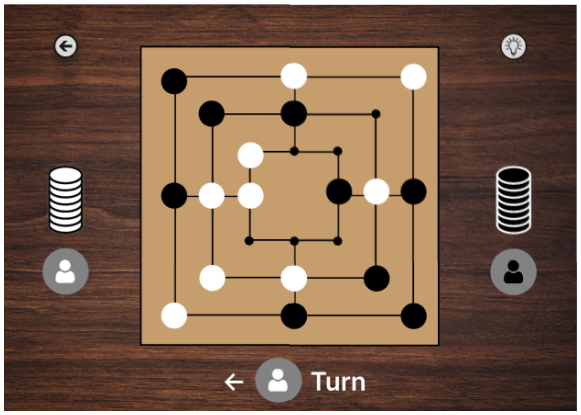
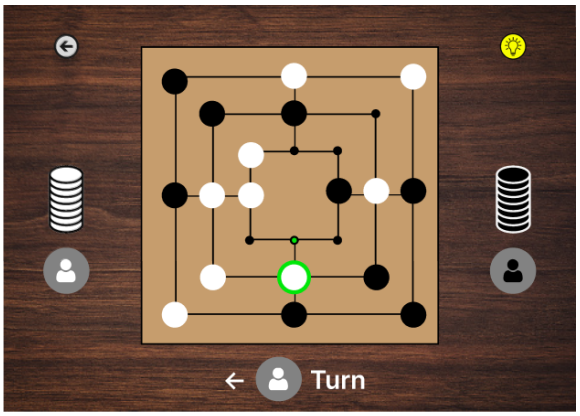


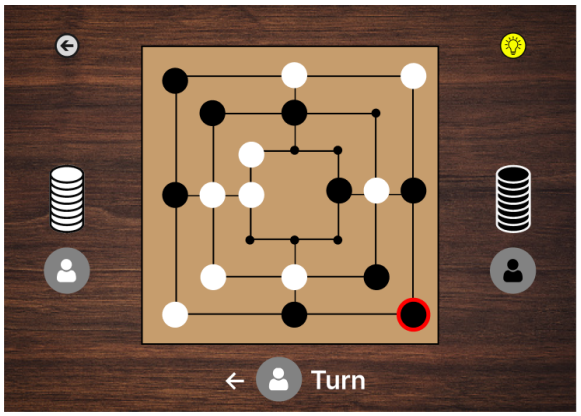
When players want to make an action (move, fly) with a placed piece, the selected piece will highlight as green. If a player has hints on, valid positions will be highlighted, see below. Once it was selected, the player just needed to click on the preferred position. The piece will only move there if it is valid.

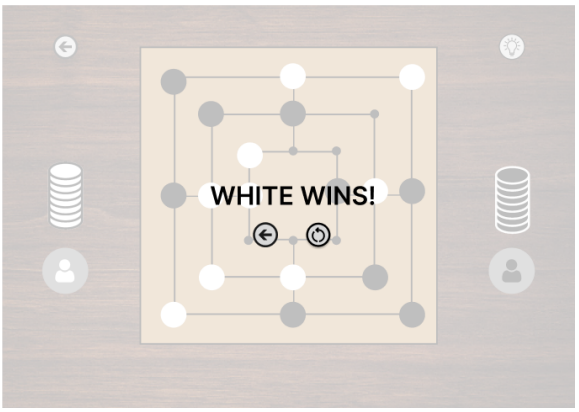
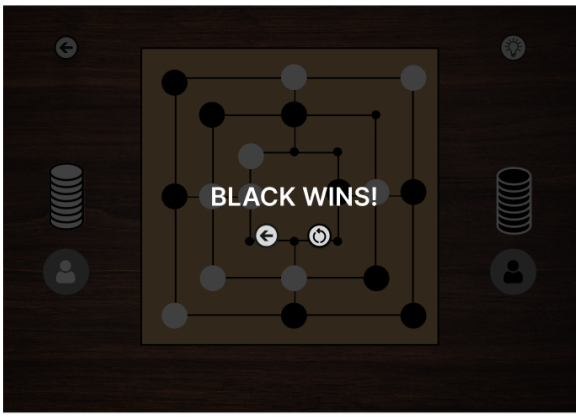
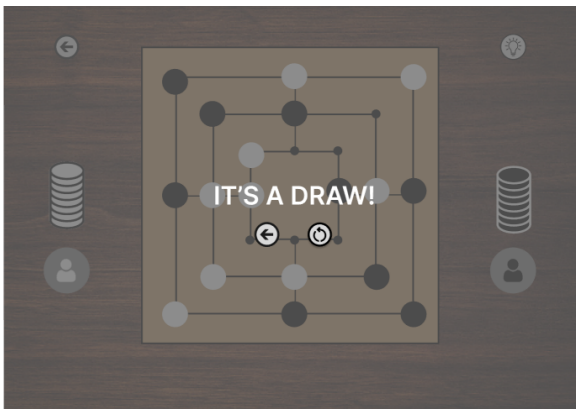
Justification:
This highlight should not be a hint since the player needs to know which one is selected. In the real world, we could see a person's hand picking a piece. But on a computer, we need to do it the digital way. Otherwise the player will be confused since they expected feedback. The colour green is chosen because green usually means OK, or good in our society. With this green highlight as feedback, the player will understand their action is valid.

Making a mill	
	When three of the pieces are aligned, those pieces will have a yellow circle surrounding them and a line connecting them to indicate their relation as a mill. All mills will be highlighted as so. We define “Hints” as the tips that are related to player moves. Therefore, the mill will be still highlighted. **Justification:** We need another sharp colour to indicate a mill. Yellow is usually used to grab your attention like a warning. It suits our needs.

Selecting a piece of opponent’s board to remove after creating a mill	
	The black pieces that could be removed will have a green circle surrounding them to indicate they are valid moves. This highlight will only show after a mill is created and could be removed. The bottom suggestive text will signal that player action. Here, it will be changed to suggest the player to remove one black token. **Justification:** A piece needs to be removed when a mill has been created. This action is mandatory, so that it has to be highlighted to remind the player to remove one of the opponents pieces. Otherwise the game cannot be continued.

Hints toggle	
 Hints OFF  Hints ON	When the hints button is clicked, the button will light up to indicate hints have been toggled on. With hints turned on, when the player selected a piece, the position that this piece could move to will be highlighted in the middle (as a green dot). **Justification:** We want the non-hints highlighted as minimum as possible. Therefore we only keep the essential highlights for usability (keep distractions to a minimum). Hints are made for assisting the player, therefore, by highlighting where that piece could go, the player could spend less effort figuring where it could go.

Hints ON: Selecting an invalid move	
	When the player selects a piece that could not be moved, that piece will be highlighted with a red circle to indicate so. **Justification:** We chose red because red usually means forbidden, stop, danger in our society. It could show that selecting that piece is invalid.

Result	
  	The result is in the middle and back to the menu button below as well as the play again button. **Justification:** The result page is only used to tell the user the result of the game. The user only needs to do two things at this page, check the result, and exit this screen. Upon exiting this screen, we got two expectations: the user wants to try this mode again, or switch to another mode. Therefore we got a back to menu button for switching modes, and a play again button for playing again the same mode while reducing the button to click or return to the menu first.

~ End ~