FIT 3077
Semester 1

Team Information

Torino Development United:
Soo Guan Yin, Chua Jun Jie, Justin Chuah, Lim Fluoryynx

Table of Contents

Team Name

Torino Development United

Team Photo

Team Membership

Name	Contact Details	Technical and Professional Strength	Fun Fact
Chua Jun Jie	**Email**: jchu0057@student.monash.edu **Phone**: 0177338896	Comfortable using Python, Java, Javascript and HTML. Can understand things in an easy and fast way.	Admitted to hospital after eating 5 fried chicken breasts.
Soo Guan Yin	**Email**: gsoo0005@student.monash.edu **Phone**: 0135339558	Comfortable with using python, JavaScript and HTML. Having more experience with backend development.	I like coffee. Admitted to hospital after eating Korean spicy noodles.
Justin Chuah	**Email**: jchu0056@student.monash.edu **Phone**: 0122895924	Comfortable with Python and Java. Able to pick up new concepts relatively well.	I share the same day of birth with one of the US presidents.
Lim Fluoryynx	**Email**: flim0012@student.monash.edu **Phone**: 0186695811	Comfortable with Python, JavaScript and HTML. Able to pick up new concepts well.	I am not fun

Team Schedule

A scheduled meeting will be held on every Friday of the week. The purpose of the meeting is for each other to check up on another's progress. If any team member is facing difficulties, he/she should voice out the difficulties during the meeting.

There also exists a team wiki document that is in Gitlab. The wiki is used for each team member to record their individual progression on the tasks they are working on. It consists of information such as task name, task assignee, total working hours etc. The wiki is updated regularly, at least once per week from the team. Furthermore, the team uses Trello for task tracking and monitoring. When the team member obtains a task, he/she should add the tasks into the Trello as well.

There is no fixed work schedule for the team members, but each team member is expected to make some progression each week.

Technology Stack and Justification

Potential Language/Framework for Frontend

ReactJS (JS + HTML)

ReactJS is a robust platform used for building web applications. It comes with an extensive built-in library containing a diverse range of features that can be intricate and challenging to implement. However, this allows developers to save time in creating their applications as they don't have to create these features themselves. One of the drawbacks of using web-based applications is that the logic or backend of the application must be implemented in JavaScript, which doesn't comply with the requirements of the assignment as it is not an object-oriented programming language. A limitation of developing a web-based application is that it necessitates the installation of a browser on the client's device to access and run the application.

JavaFX

JavaFX is a collection of graphics packages in Java that enables developers to build client-side applications. Its user interface is user-friendly and facilitates the creation of the application's user interface with ease. A significant advantage of using JavaFX to develop the application is that the application's logic or backend can be developed in Java, which satisfies the assignment's requirement of utilising an object-oriented programming language.

Potential Language/Framework for Backend

Python

Python is a versatile programming language that can be used for a wide range of tasks involving programming. It can be used to create both frontend UI and backend logic. All members of the team possess a strong knowledge of the language and are comfortable using it to develop applications. However, it doesn't completely meet the assignment requirement of utilising an object-oriented programming language. This is due to the fact that while Python only supports all the concepts of object-oriented programming, it is possible to write code without creating a class.

Java

Java is a programming language that adheres strictly to the principles of object-oriented programming. Concepts such as encapsulation, abstraction, and inheritance from object-oriented programming help to safeguard the application against unwanted data access and function manipulation by the user. This enhances the security of the application. Finally, all team members have adequate knowledge of the language, which is sufficient to create a smaller scale application.

Final Decision

From the comparison above, the team had chosen Java as the backend language to build the core logic of the application as Java adheres strictly to the principles of object-oriented programming. As for the creation of GUI of the application, the team believes that using JavaFX will decrease the time needed to develop the GUI of the application. Furthermore, the process of combining the frontend and backend of the application does not need to include any “middle man”, as both front and back end of the application uses the same language.

Discarded Alternative

The decision to not use ReactJS as the frontend framework is based on the requirement for an external browser application to view and interact with the application. Additionally, when developing the backend logic with Java, an API must be used to communicate with the frontend side of the application.

Python is not being used as the backend language for developing the logic due to its incapability to fully follow the rules and concepts of object-oriented programming, which does not satisfy the assignment requirement of using object-oriented programming.

FIT 3077
Semester 1

User Stories

Torino Development United
Soo Guan Yin, Chua Jun Jie, Justin Chuah, Lim Fluoryynx

Table of Contents

User Stories

Basic Requirement

User Stories 1 (Playing against friends with the same device)

As a player, I want to play against another player on the same device so that the game can be played in real time.

User Stories 2 (Placing token on the board)

As a player, I want to be able to place a token on the board, so that I can make my move.

User Stories 3 (Removing opponents token from the board)

As a developer, I want the player who created a mill to remove a token on the board, so that the player can gain an advantage.

User Stories 4 (Sliding token along the board line)

As a player, I would like to slide my token along the board line to any empty adjacent intersection so that I can end my turn.

User Stories 5 ("Flying" token)

As a developer, I want tokens to become flying tokens when a player only has 3 tokens left so that I can gain an advantage.

User Stories 6 (Quit game)

As a player, I want to exit to the main menu during the game so that I can stop playing whenever I want to.

User Stories 7 (Notify turns)

As a player, I want to be able to see whose turn it is every time after a move is made, so that I can keep track of the game progress.

User Stories 8 (How many tokens left to place)

As a player, I want to see the remaining number of tokens I can place, so that I can plan my moves.

User Stories 9 (Alternating turns)

As a player, I want to be able to move once my opponent has finished making a move so that I can take my turn.

User Stories 10 (Can't break mills)

As a player, I want tokens in a mill I created not be able to be removed so that I can gain an advantage.

User Stories 11 (Remove token cannot return)

As a developer, I want tokens that are removed to be permanently removed so that the player who removed the token can gain an advantage.

User Stories 12 (Player who has no legal moves left loses)

As a developer, I want the player who has no legal moves remaining on board to lose the game so that the game is completed.

User Stories 13 (Player who has two tokens left loses)

As a developer, I want the player who has two tokens remaining on board to lose the game so that the game is completed.

User Stories 14 (Select a token to move)

As a player, I want to be able to select a token to move so that I can move the token I want.

User Stories 15 (Game is drawn)

As a developer, I want the game to draw when both players have only three pieces left so that the game can conclude.

User Stories 16 (No illegal rules)

As a developer, I want all players to be restricted from performing illegal moves, so that all players play at an even playing field.

User Stories 17 (Flying token)

As a developer, I want flying tokens to move to any empty intersection on the board so that the player has more options.

User Stories 18 (Select a side)

As a player, I want to select heads or tails so that I can flip a coin to decide who goes first.

User Stories 19 (Flip a coin)

As a developer, I want players to toss a coin to decide who will go first, so that the game can start.

User Stories 20 (Colour that go first)

As a developer, I want the player who goes first to use white tokens so that the game can start.

Additional Requirement

User Stories 21 (Tutorial)

As a new player, I want a tutorial mode so that I can learn how to play the game.

User Stories 22 (Play with computer)

As a player, I want to be able to play with the computer, so that I can play the game when I am alone.

User Stories 23 (Hint)

As a beginner player, I want to be provided with hints so that I can know what legal moves can be made.

User Stories 24 (Hint)

As a player, I would like to have a button to toggle the hint option, so that I can play the game without assistance.

User Stories 25 (Choose game mode)

As a developer, I want the available game modes to be presented to the player so that they can choose their desired game mode.

FIT 3077
Semester 1

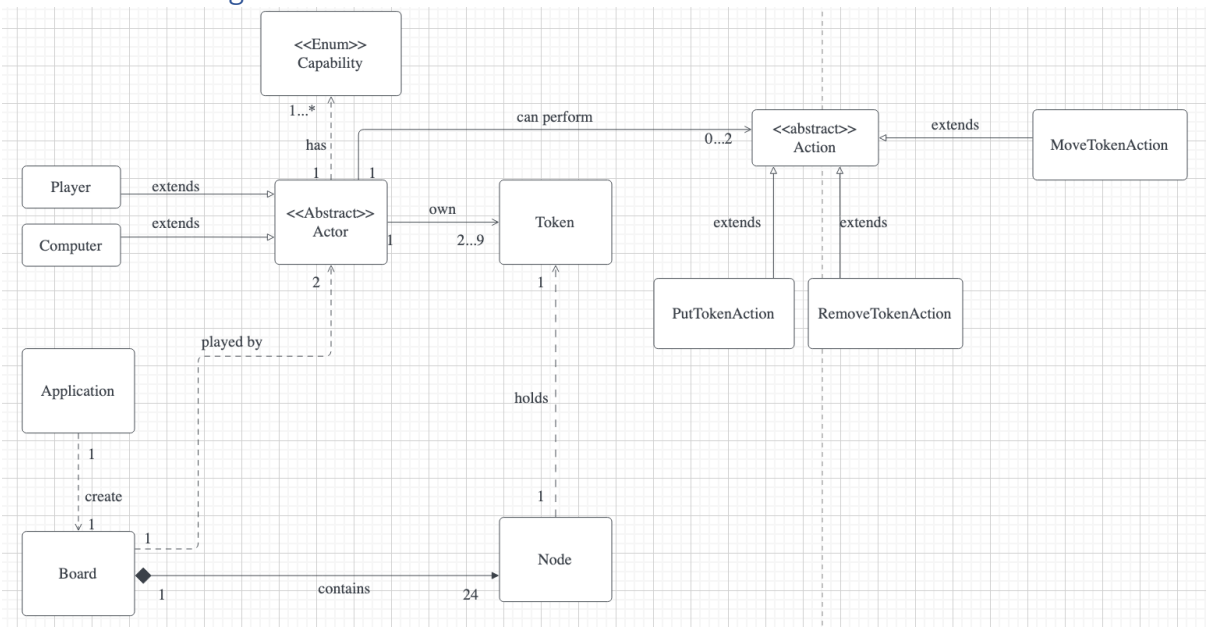
Basic Architecture & UI Design

Torino Development United
Soo Guan Yin, Chua Jun Jie, Justin Chuah, Lim Fluoryynx

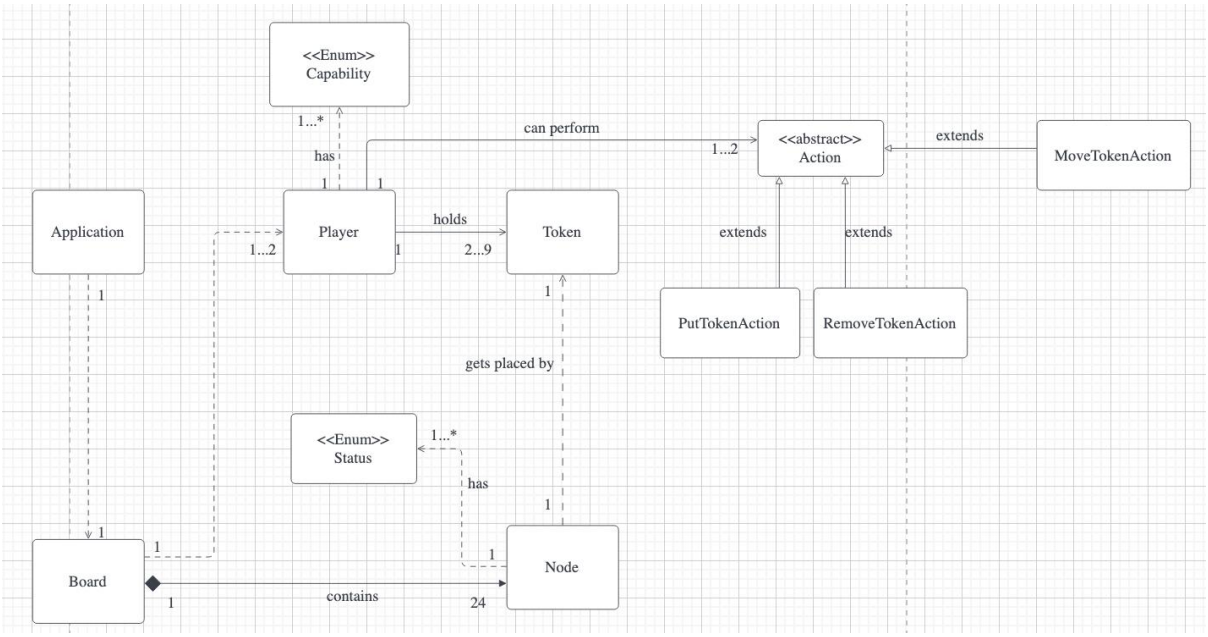
Table of Contents

Basic Architecture

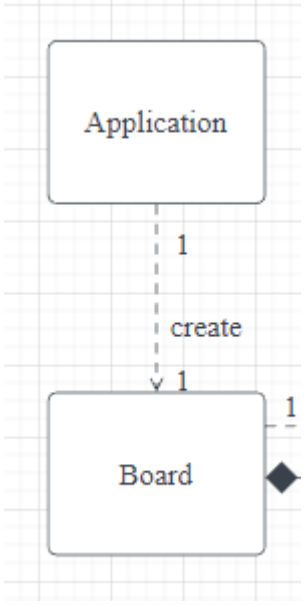
Chosen class diagram



Discarded alternatives



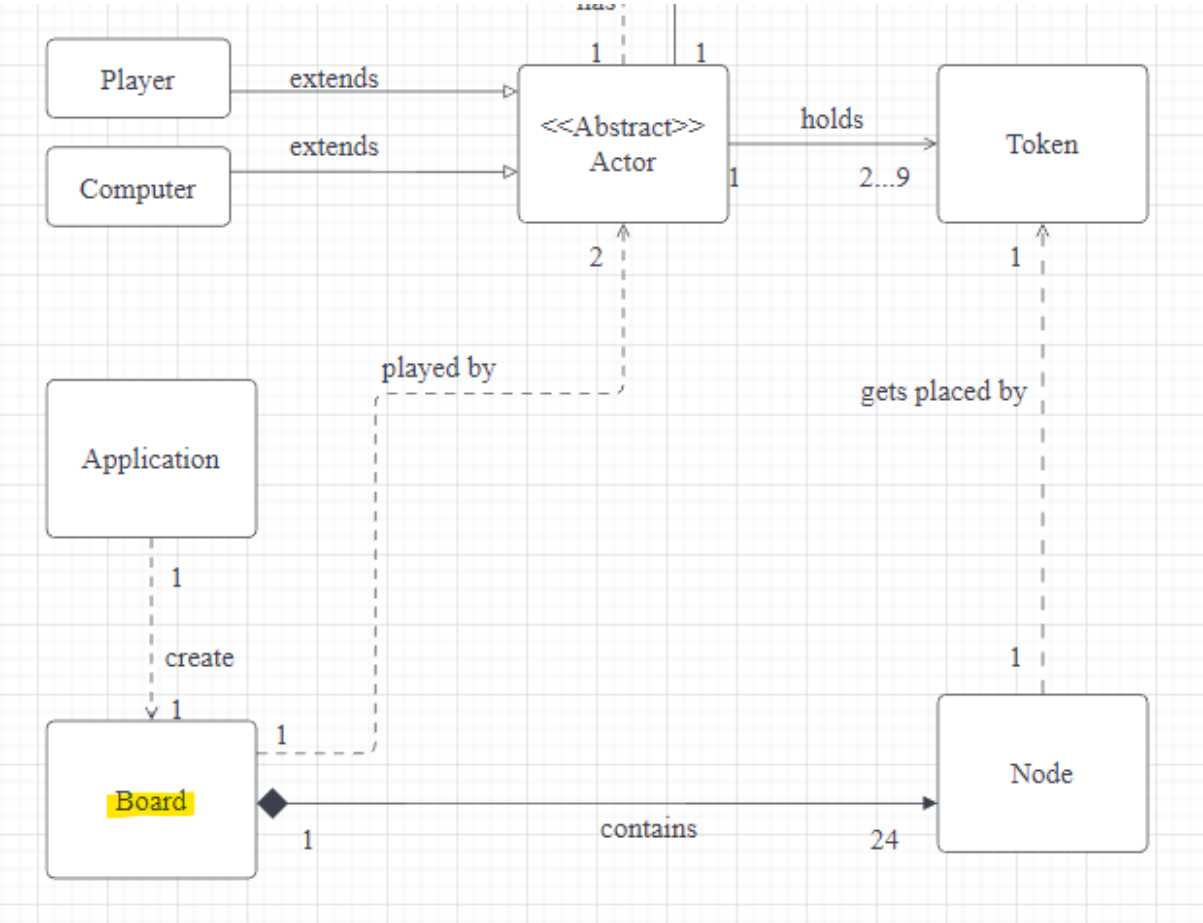
Class “Application”



From the figure above, Application class is the class that the user first interacts with to be able to play the game. For now, we assume that this class will be a relatively small class compared to the other classes, but this is to adhere to the Single Responsibility Principle (SRP). Users will be able to select the type of game mode through this application, which would then call a one-off method to initialise the game board, represented as the Board class. This also explains the dependency relationship between these two classes and the corresponding multiplicity as there can only be one application and one board at a time. Another reason we have decided to create the Application class as well is because we eventually plan to implement a dedicated user interface where the logic will also fall under this particular class.

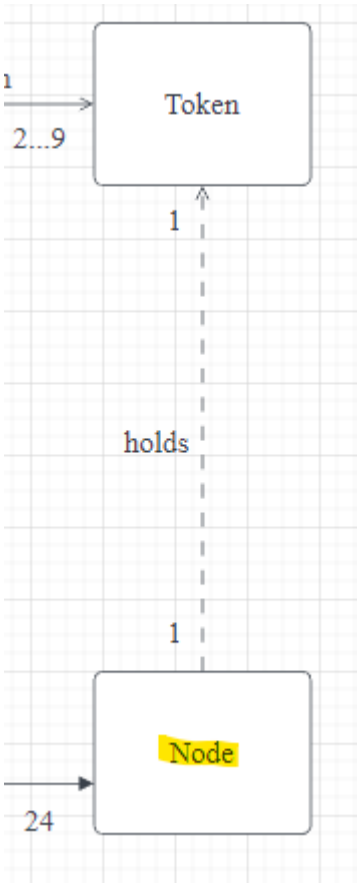
A design that was initially proposed was to have a dependency relationship between the Application and the Player class, as we had assumed that the Application was needed to initialise all the players. However, this was then discarded as this would overload the number of responsibilities under the Application class violating the SRP and eventually becoming a “God” class when we implement the user interface for the game.

Class “Board”

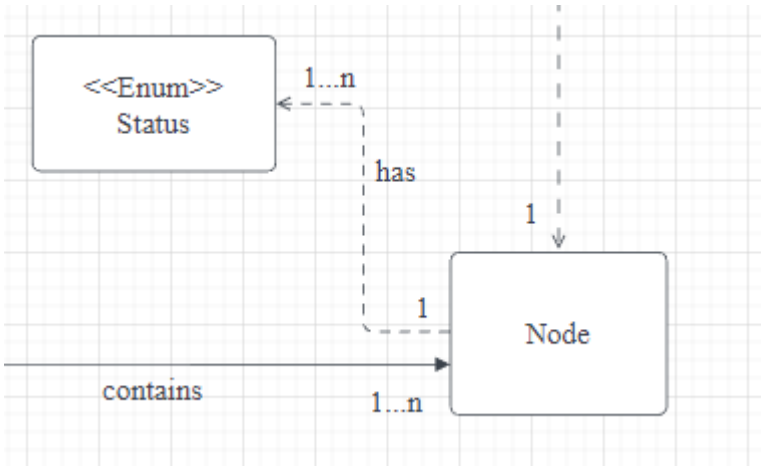


As shown in the figure above, the Board class will be representing the game board in which the players will be using to play the game. There is an aggregation relationship between the Board class and the Node class as our team has decided that if there is no game board, there will be no nodes for tokens to be placed on the board. Since there are 24 possible locations for tokens to be placed on a standard board, the multiplicity is presented as such. Since the game can be played in a “Player vs Player” format where there will be 2 real players, Tutorial format and “Player vs Computer” format where there is 1 real player and 1 computer, the multiplicity relationship between the Board class and Actor abstract class is presented as shown. At the end of every player move, and at the start of the game before the first player makes his turn, we plan for there to be a ticking mechanism that ticks the board to calculate all the possible legal moves that can be allowed for the actors which explains the dependency relationship.

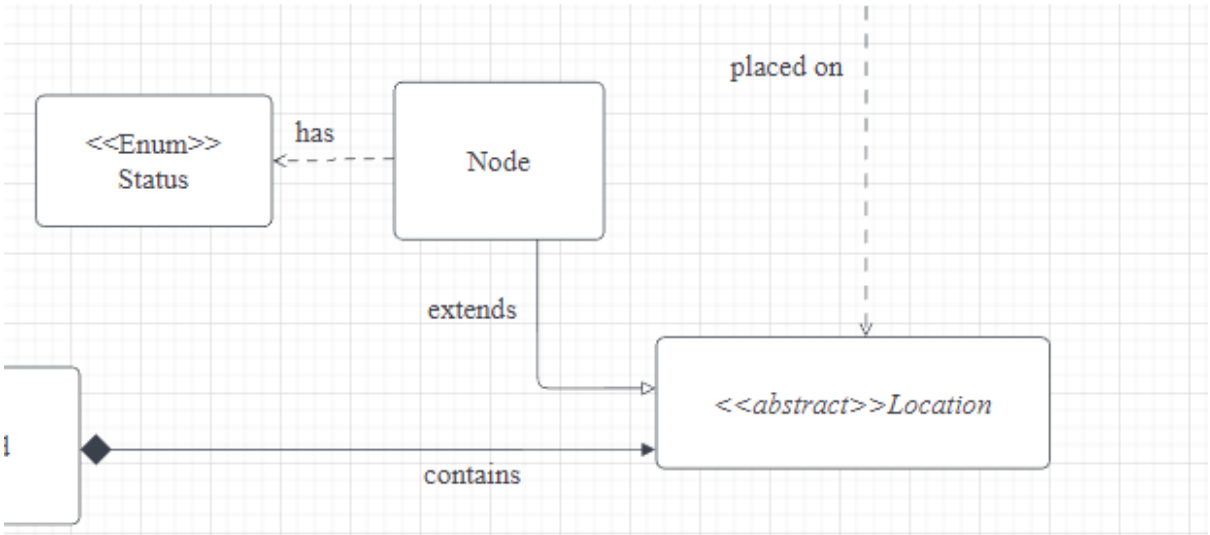
Class “Node”



As shown in the figure above, the Node class will be representing the areas in which tokens can be placed. Nodes will be storing the instances of Token, and since only one token can be on one node at a time, this explains the dependency relationship and multiplicity as shown.

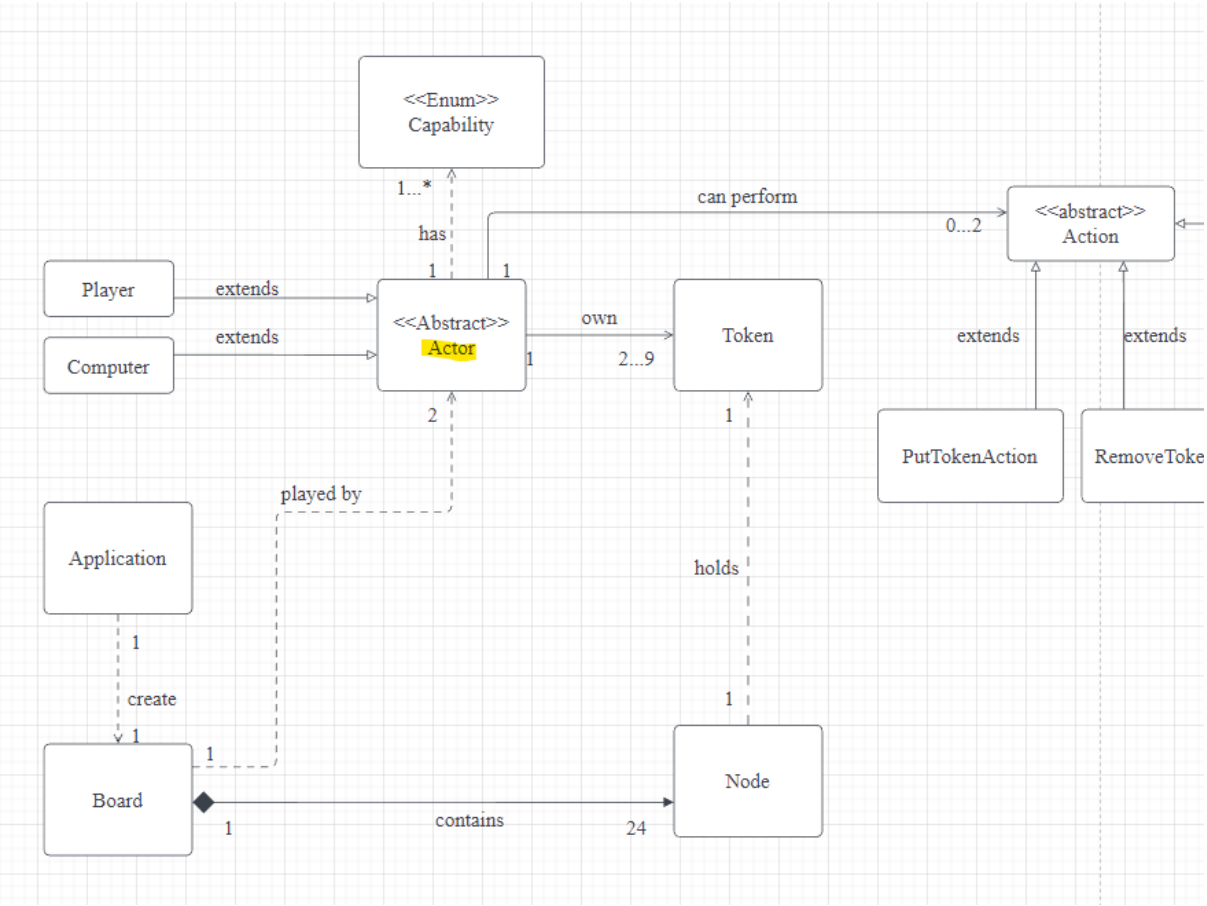


An early proposed design was to have an enumeration to represent the status of the node. This would be done by assigning a constant Status to a Node to signify that the node has a token on it. This idea was then discarded as we realised that it can be delegated to the Board class as a method, which will be called simultaneously to calculate the legal moves that can be made by the next player.

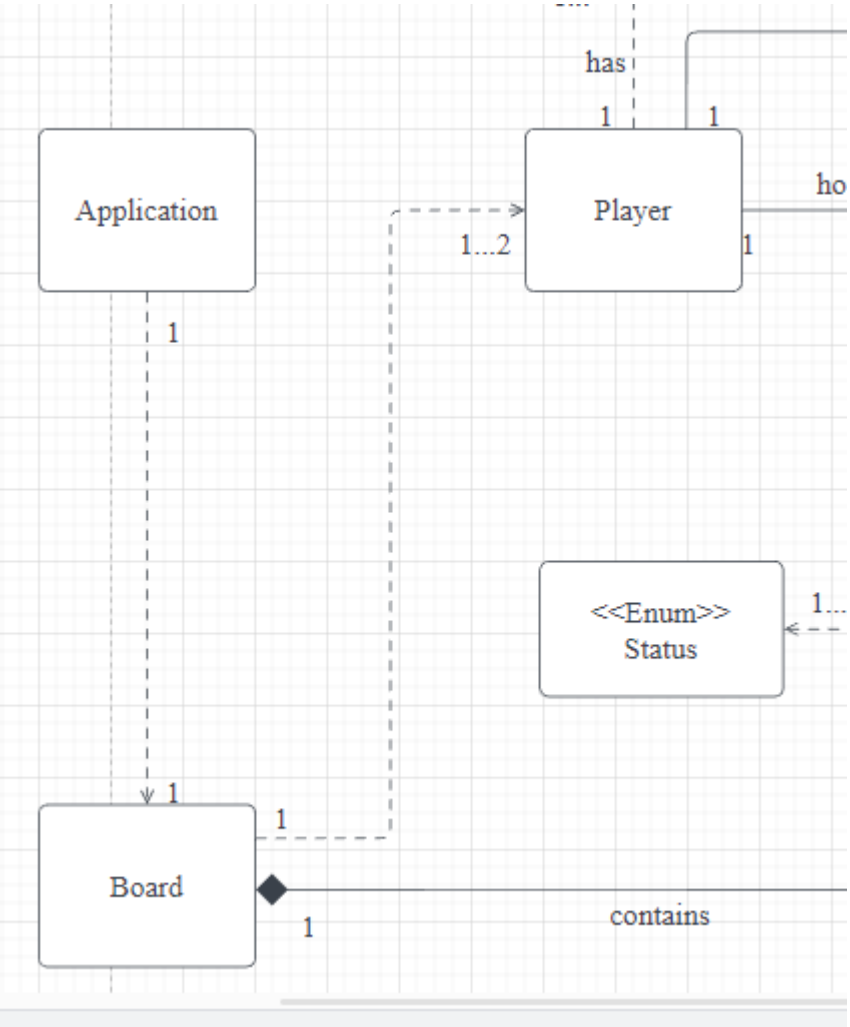


Another design that was proposed as well was to use an abstract class called Location and Node were to extend from it. But after thorough discussion we have decided that was going to be unnecessary based on the current requirements and the additional requirements provided since they would all be using Nodes and that the attributes of Nodes would not change. Thus, we have decided to omit the creation of an extra abstract class and obvious differences can be seen aside from a reduction of classes in the domain modelling.

Abstract Class “Actor”

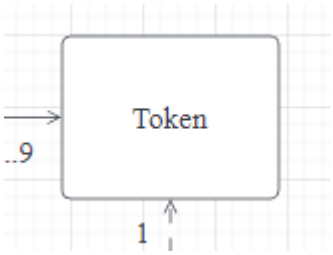


As shown in the figure above, an abstract class Actor was created, and the Player and Computer class extends from Actor. This is because we as a team have made a common assumption that if the computer class were to just be a random move selector as per the suggested manner of implementing the computer, it functions similarly to a normal player in terms of what can be done, except that this time it is random. We decided to take into consideration the Dependency Inversion Principle (DIP) so that the heuristic of the computer class can be improved on from just being a random move selector if there is a need to in the future. The enumeration Capability is to prevent the Player of Computer from playing any illegal actions, and since an Actor can select an action from a list of possible actions, there is 1…* multiplicity as shown in the figure. As per the rules, an actor will store a list of actions that can be made, and is capable of performing 2 actions in a turn if a mill were to be formed and an opponent’s token can be removed, 1 action if the player can only put a token or move a token and if the actor is incapable of making any actions he has lost the game, hence explaining the association relationship and multiplicity . An actor will have an inventory list consisting of tokens, in which he can own a maximum of 9 tokens at a time down to a minimum of 2 tokens at a time, hence the association and multiplicity as shown. A design choice was also made here in whether we were to decide on the Actor to Action multiplicity to be a 1 to 1…2 or a 1 to 0…2 relationship. We decided with the latter as we collectively decided to use an enumeration to update the action list for when the actor has no legal moves available, thus not having any legal actions to select from. This is just to adhere to the SRP as the enumeration Capable was designed solely to show what actions can be taken by the Actor.



A design that was originally proposed was to combine both the Computer and Player under one Player class since we originally assumed them to be doing the same things, just with a simple condition that proposes a random generator to determine the action to be made if there was a flag that would identify this Player as a Computer. However, upon further consideration, we realised that this would violate the Open-Closed Principle as if we were to improve the set of heuristics of the computer, we would then need to implement changes directly to the Player class.

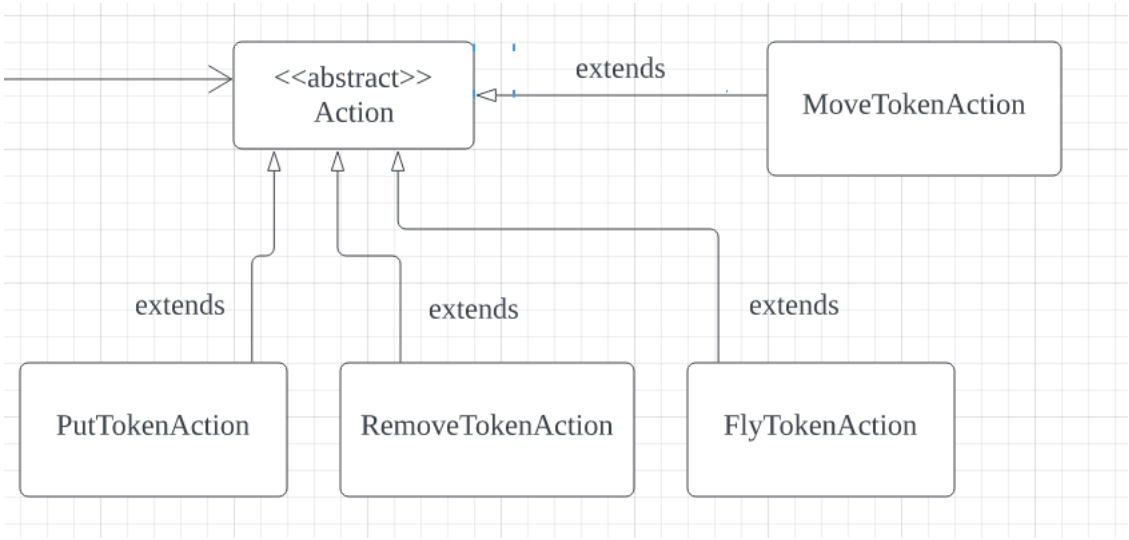
Class “Token”



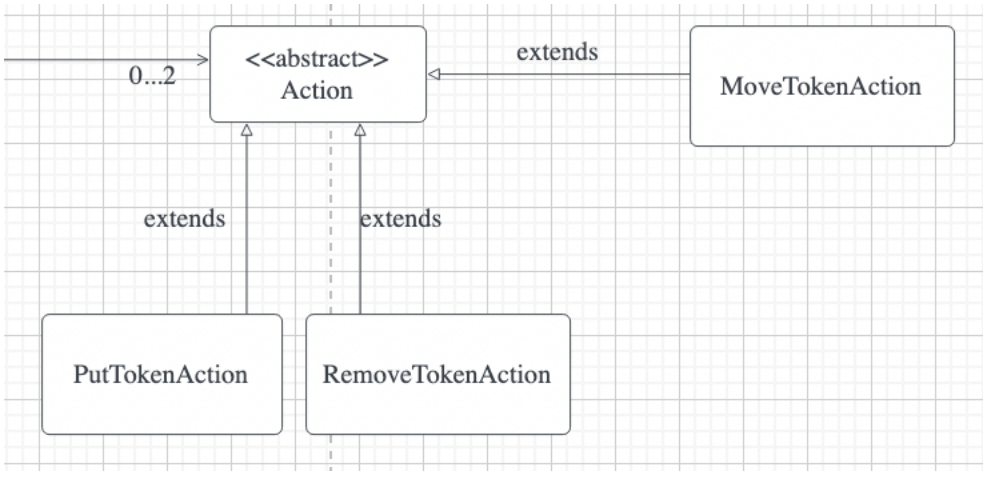
The Token class represents the “Men”. This class was created with the idea of Single Responsibility Principle in mind as we would use this class to store attributes of the token such as colour to show whose token this belongs to and does it belong in a mill.

Action abstract class

Discarded alternative



Chosen domain



Action class will be representing possible types of moves that can be performed by actors involved in the game. It is an abstract class created to be the parent class of PutTokenAction,

RemoveTokenAction and MoveTokenAction class as those classes share some same methods and attributes.

This Action abstract class is created to reduce duplication of code, and avoid breaching the Don't Repeat Yourself (DRY) principle as commonly used methods do not need to be copy pasted in every subclass.

Action class will be mainly responsible for updating the number of tokens in Actors 'token list or Actors 'token on the board.

Actions that can be performed by actors are separated into subclasses in order to adhere to the Single Responsibility Principle (SRP) as every subclass will only be responsible for a single type of movement (as described in sections below).

Furthermore, by extending action subclasses from Action class, Open Closed Principle (OCP) can be applied as the class can be more extensible for new actions to be added during future developments without modifying existing code such as association from the Actor class. Without an Action abstract class, the Actor class will need to have association to more classes.

PutTokenAction

This subclass will contain methods to check for empty nodes on the board, subtract the token from the actor's token list, and allow the actor to place the token on an empty node.

MoveTokenAction

This subclass will contain methods to check for empty adjacent nodes on the board, and allow the actor to slide the token to an adjacent empty node.

RemoveTokenAction

This subclass will contain methods to check for an opponent's token that is not part of a mill, allow the actor to remove the token if it is not part of a mill, and subtract the amount of opponent's token on the board.

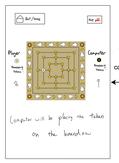
FlyTokenAction (discarded)

This subclass is created in early proposed design and was discarded afterwards because the behaviour of tokens in FlyTokenAction and PutTokenAction are the same, in which a token is allowed to be placed at any empty nodes on the board. The only difference is that FlyTokenAction is to be performed by Actor that is left with three tokens on the board, while PutTokenAction is to be performed by Actor when Actor still has tokens in their token list.

Hence, we decided to discard this class and allow Actor to perform PutTokenAction when Actor still has tokens in their token list or is left with three tokens on the board. This can be done by checking and updating the number of tokens players have in the token list and on the board after every turn.

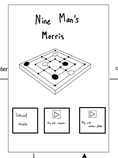
Basic UI Design

Full image is a below.

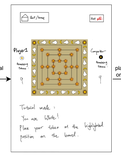


When player finish his/her step, the computer will proceed to make a move

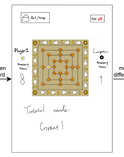
When user click play with computer



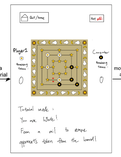
click tutorial mode



place a token on the board

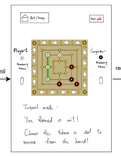


moves to a different tutorial

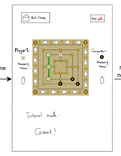


moves the token accordingly

formed a mil

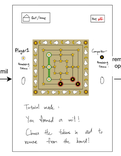


removed one token

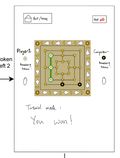


move to a new tutorial

formed a mil



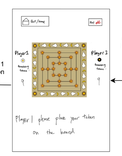
removed a token opponent left 2 tokens



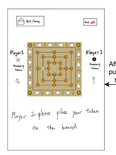
goes back to main menu

when user Click play with another player

When the coin finish flipping and result is displayed both player



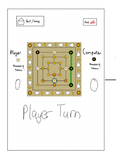
After player 1 puts token on the board



Player Selects Token to Move

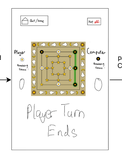
Player moves selected token up

Player does not get the option to remove opponent's tokens since they are part of the mill. Player Turn End



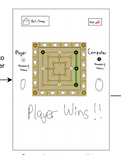
Player Selects Token to Move

Player moves selected token down

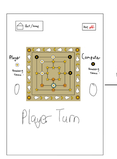


Player Turn Ends

Turn passes to Computer

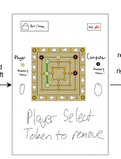


Game detects no possible moves can be made. Computer loses. Game Over.



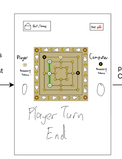
Player Selects Token to Move

Player moves selected token left



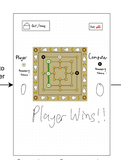
Player forms a mill, provided option to remove opponent token

Player removes upper rightmost token

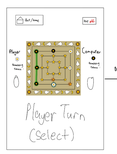


No more moves can be made. Player Turn End

Turn passes to Computer

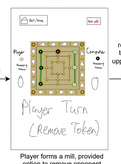


Game detects Computer only has two tokens remaining. Computer loses. Game Over



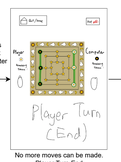
Player Selects Token to Move

Player moves selected token right

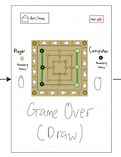


Player forms a mill, provided option to remove opponent token

Player removes token at upper center part of board

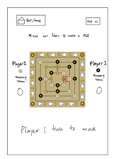


No more moves can be made. Player Turn End

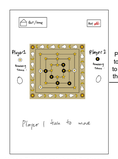


Game detects both players only have 3 tokens remaining. Game is Drawn. Game Over.

Player toggle hint to "on"

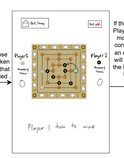


Player toggle hint to "off"

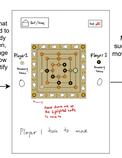


When Player1 only have 3 token left.

Player 1 choose to move the token to the position that the arrow pointed



If the position that Player 1 wanted to move to already contains a token, an error message will appear below the board to notify the player.



Move will be success if player move the token to highlighted position