FIT 3077
Semester 1

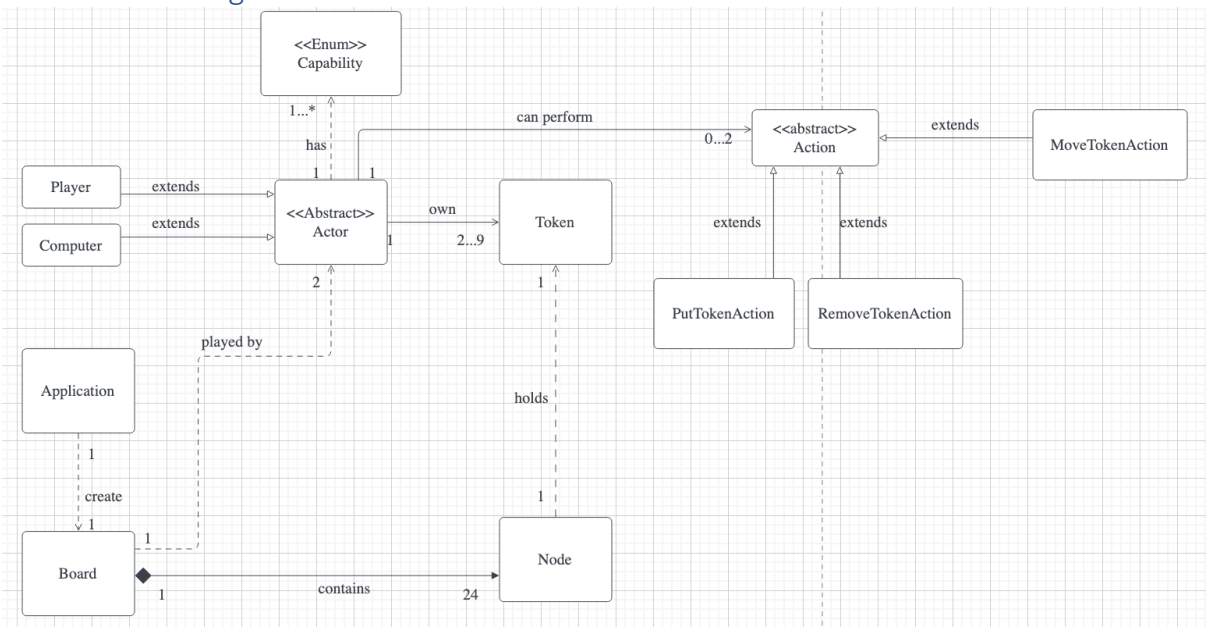
Basic Architecture & UI Design

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

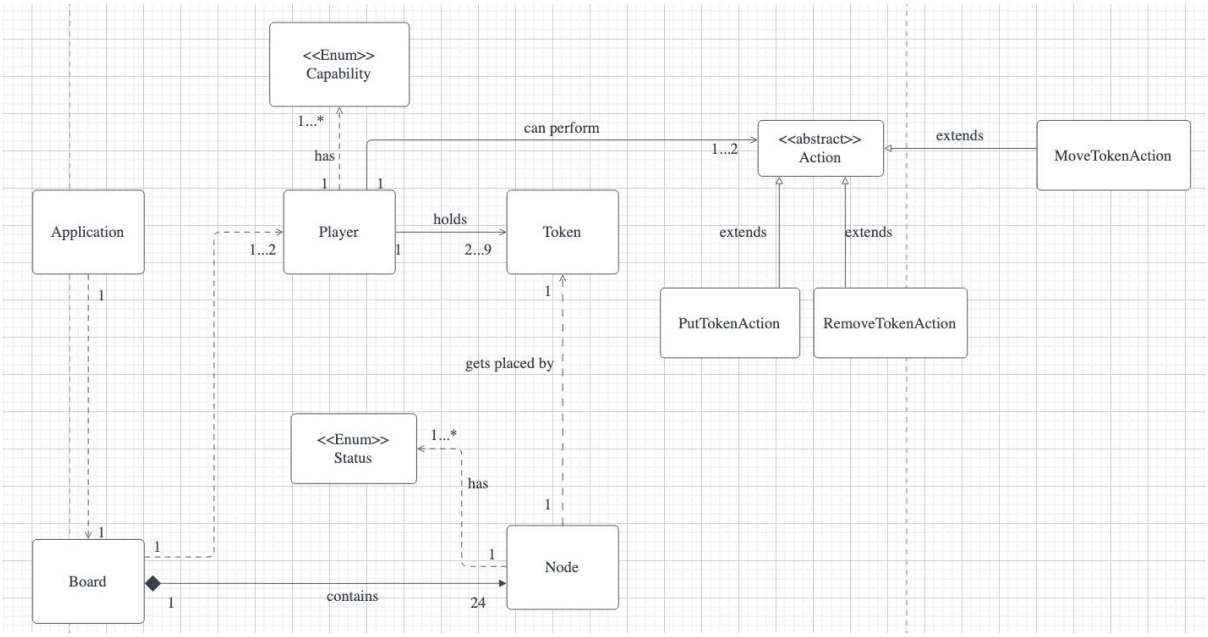
Table of Contents

Basic Architecture

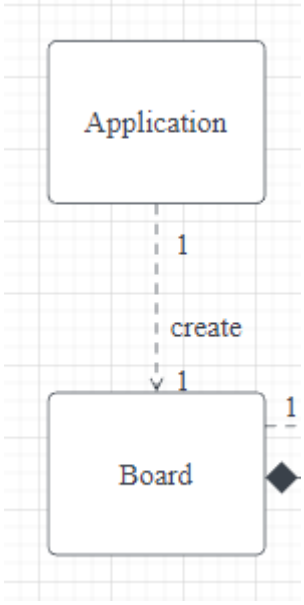
Chosen class diagram



Discarded alternatives



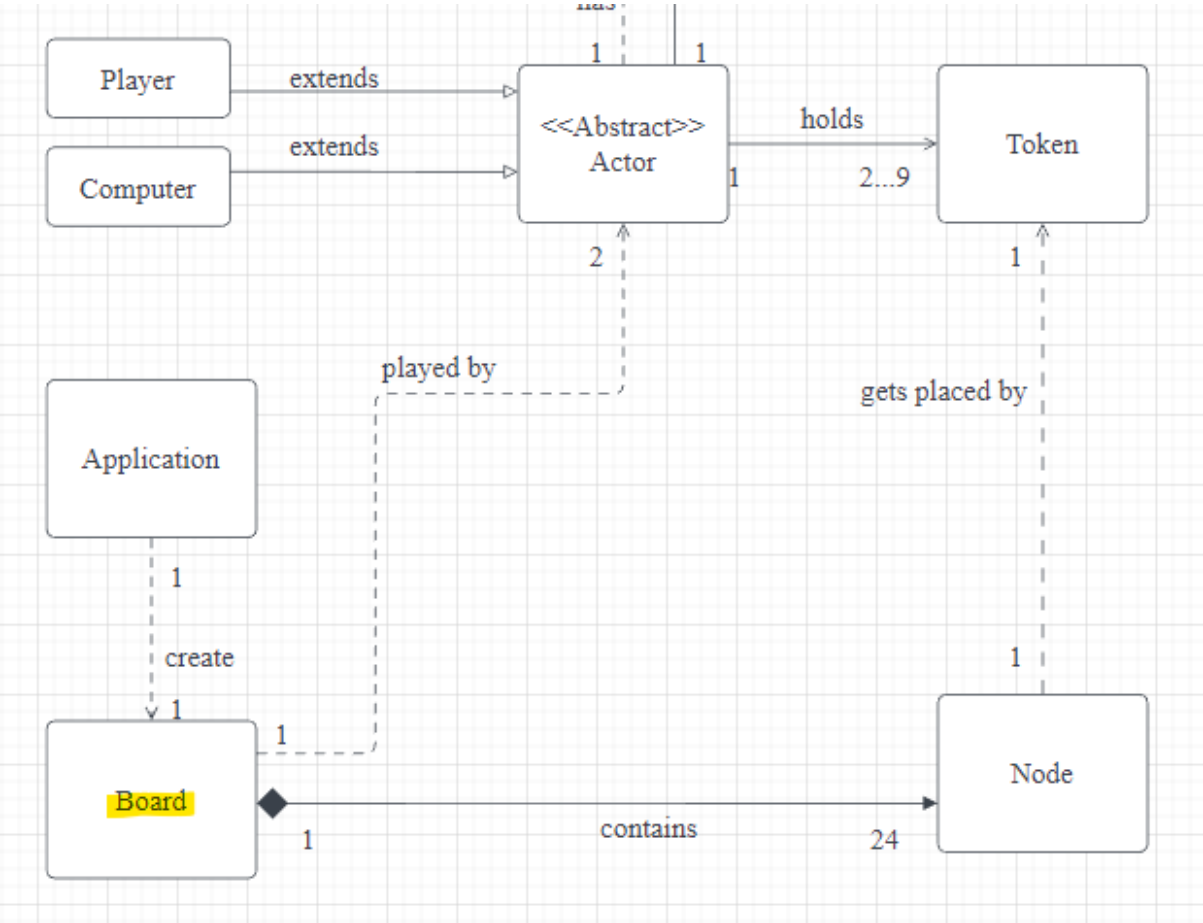
Class “Application”



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.

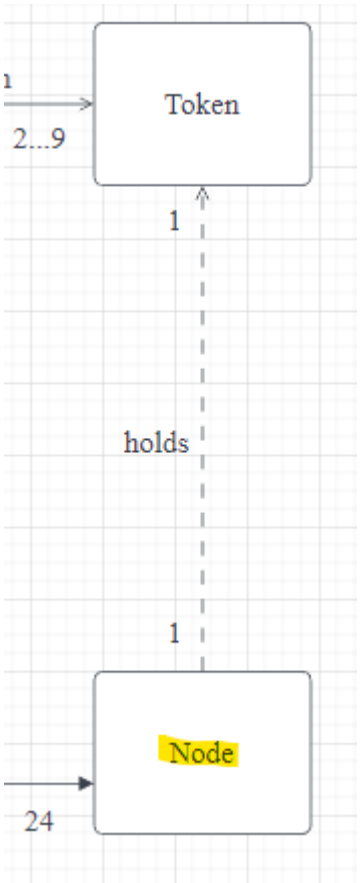
A design that was initially proposed was to have a dependency relationship between the Application and the Player class, as we had assumed that the Application was needed to initialise all the players. However, this was then discarded as this would overload the number of responsibilities under the Application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

Class “Board”

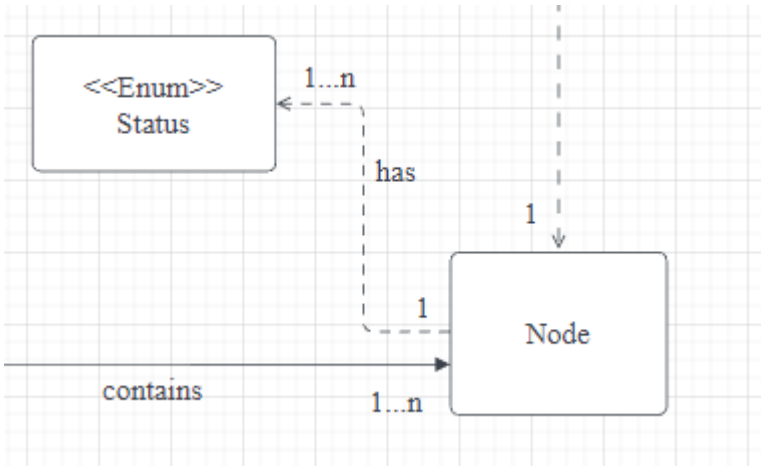


As shown in the figure above, the Board class will be representing the game board in which the players will be using to play the game. There is an aggregation relationship between the Board class and the Node class as our team has decided that if there is no game board, there will be no nodes for tokens 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.

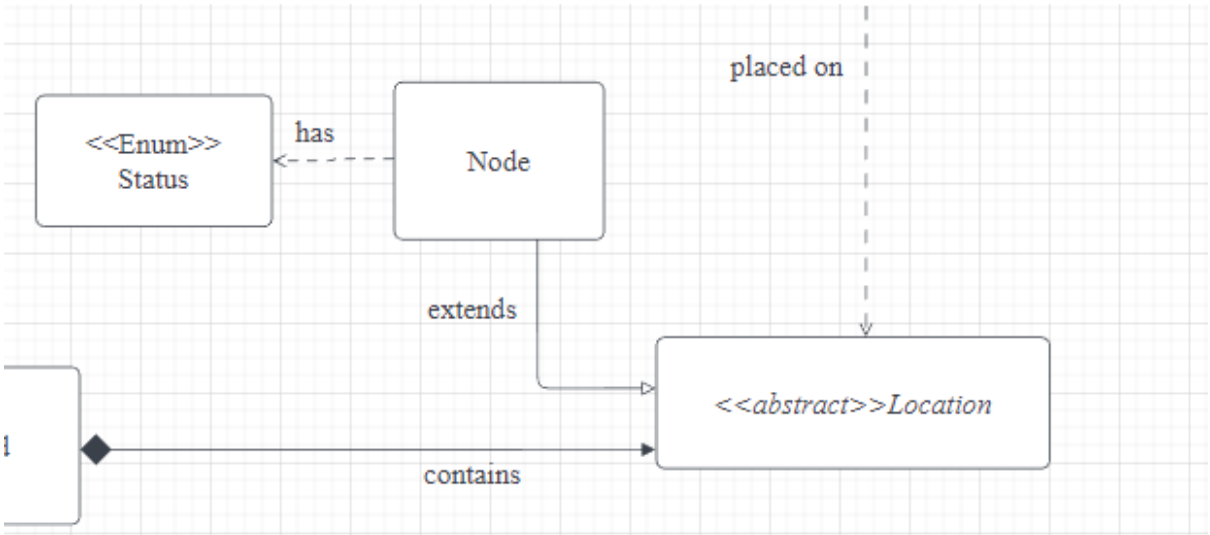
Class “Node”



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.

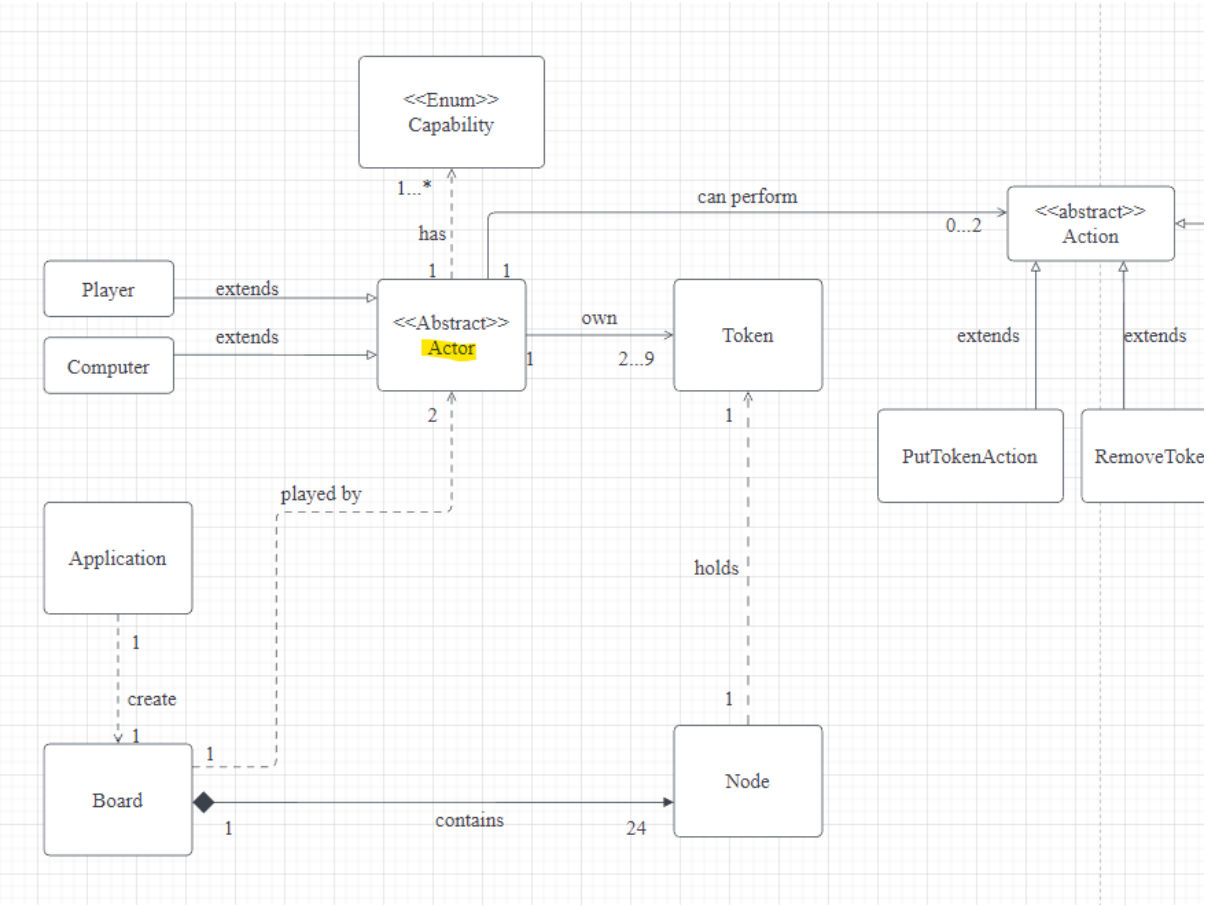


An early proposed design was to have an enumeration to represent the status of the node. This would be done by assigning a constant Status to a Node to signify that the node has a token on it. This idea was then discarded as we realised that it can be delegated to the Board class as a method, which will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

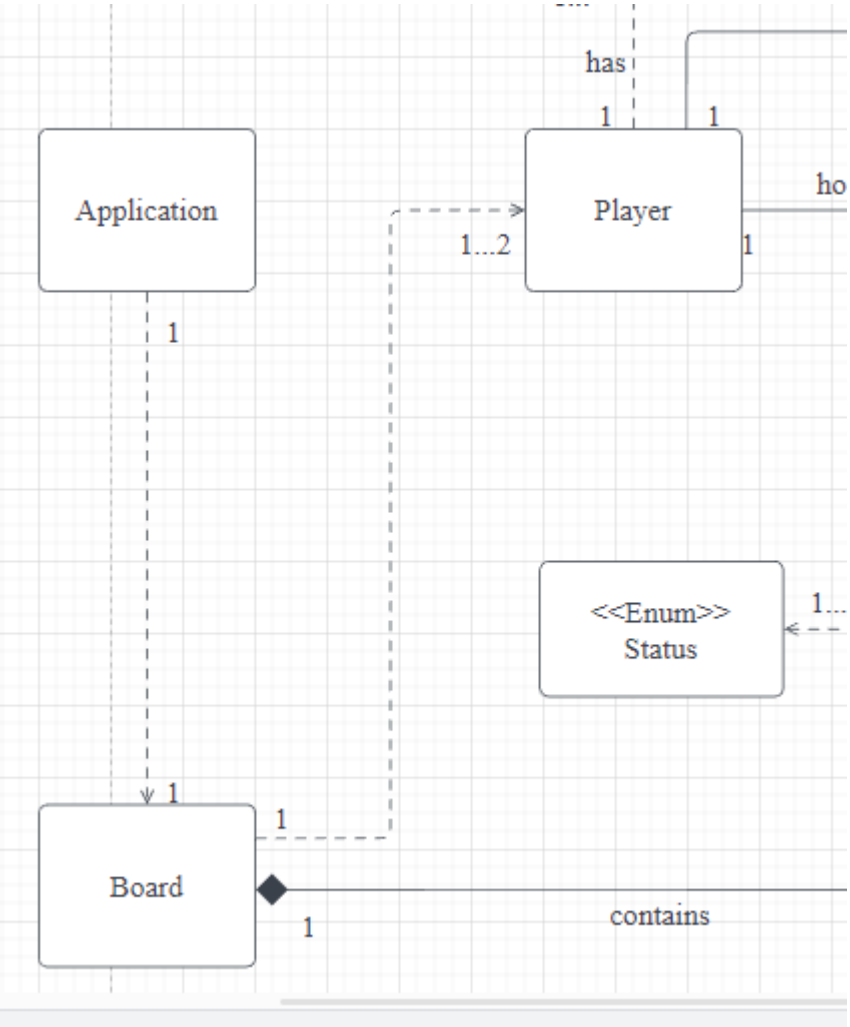


Another design that was proposed as well was to use an abstract class called Location and Node were to extend from it. But after thorough discussion we have decided that was going to be unnecessary based on the current requirements and the additional requirements provided since they would all be using 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”

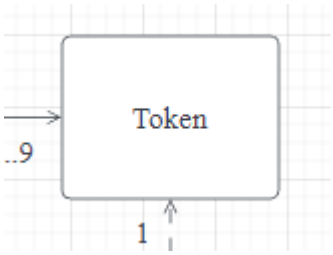


As shown in the figure above, an abstract class Actor was created, and the Player and Computer class extends from Actor. This is because we as a team have made a common assumption that if the computer class were to just be a random move selector as per the suggested manner of implementing the computer, 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.

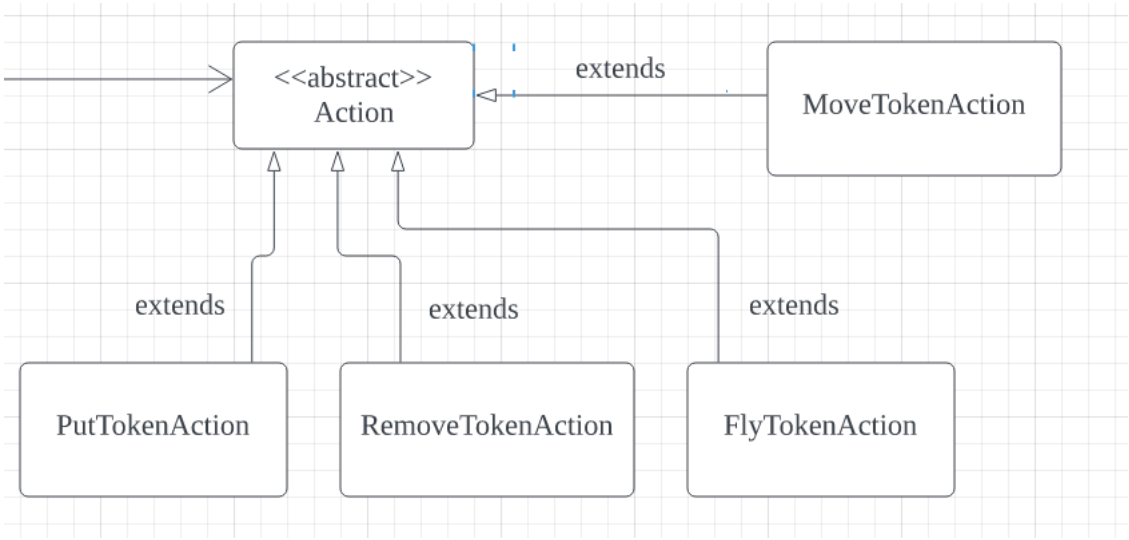
Class “Token”



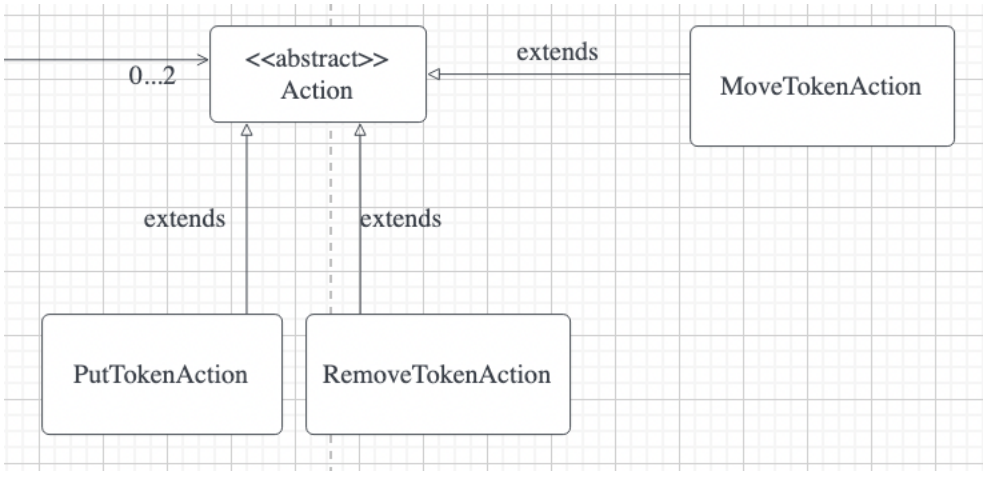
The Token class represents the “Men”. This class was created with the idea of Single Responsibility Principle in mind as we would use this class to store attributes of the token such as colour to show whose token this belongs to and does it belong in a mill.

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.