Detailed Design For Yoté
By
2 Group 2 Furious

Charles Stewart
Daniella Drew
Malka Saba
Mohamed Mohyeldin
William Hazell

Table of Contents

Executive Summary

The following being the detailed design document for the software project Yoté. Yoté is a traditional African capturing game in the Checkers family. It's played on a 5x6 board with 12 pieces per player and no standard setup, instead the players take turns placing their stones. Captures are made by jumping. A player that has just captured an opponent's piece can capture another piece for each piece jumped.

The overall design of the game is fairly simple. The game has been designed in a way that most of the entities as seen in figure 2 are individual classes. These classes include Game Manager, Board, Player, Move, YoteIO, and position. YoteIO is the main class of the program. This class essentially acts as the main control for the game. All the input from player is directed to this class and is used to communicate with the rest of the classes.

There are few limitations in the design of this game that the implementation team should be aware of. This game uses a text base interface therefore, not a lot of creativity can be implemented in the interface of this game. The second limitation is that this game will be only 2 players game. Another limitation is that even though it is a two-player game however game will be played on a single computer, this also states that the game will not be played online.

The trickiest part of the design will be implementing the move class. Majority of logic for movement on board is implemented in this class. Movements for Yote Game include: moving a piece, capturing a piece, and placing a piece.

Game Rules

(taken directly from https://en.wikipedia.org/wiki/Yoté)

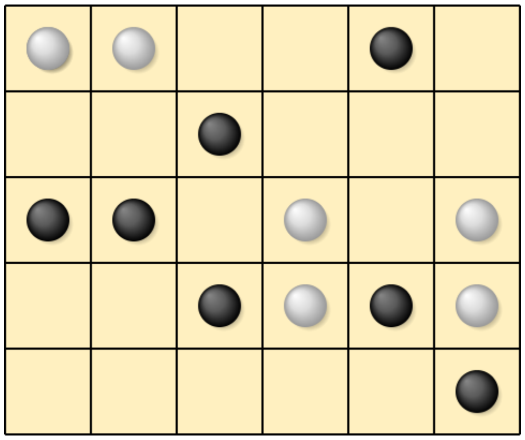

Figure 1 Yoté Game in Play

The game is played on a 5×6 board, which is empty at the beginning of the game. Each player has twelve pieces in hand. Players alternate turns, with White moving first.

Move

In a move, a player may either:

1. Place a piece in hand on any empty cell of the board.
2. Move one of their pieces already on the board orthogonally to an empty adjacent cell.
3. Capture an opponent's piece if it is orthogonally adjacent to a player's piece, by jumping to the empty cell immediately beyond it. The captured piece is removed from the board, and the capturing player removes *another* of the opponent's pieces of his choosing from the board.

Goal and End Game

The main goal of the game is to capture all the pieces of the opponent player. The game can end when either:

1. The player who captures all the opponent's pieces is the winner.
2. The game can end in a draw if both players are left with three or fewer pieces.
3. If a player to move has no move available, the game ends and the player with the greater number of pieces remaining is the winner.

Class Diagram

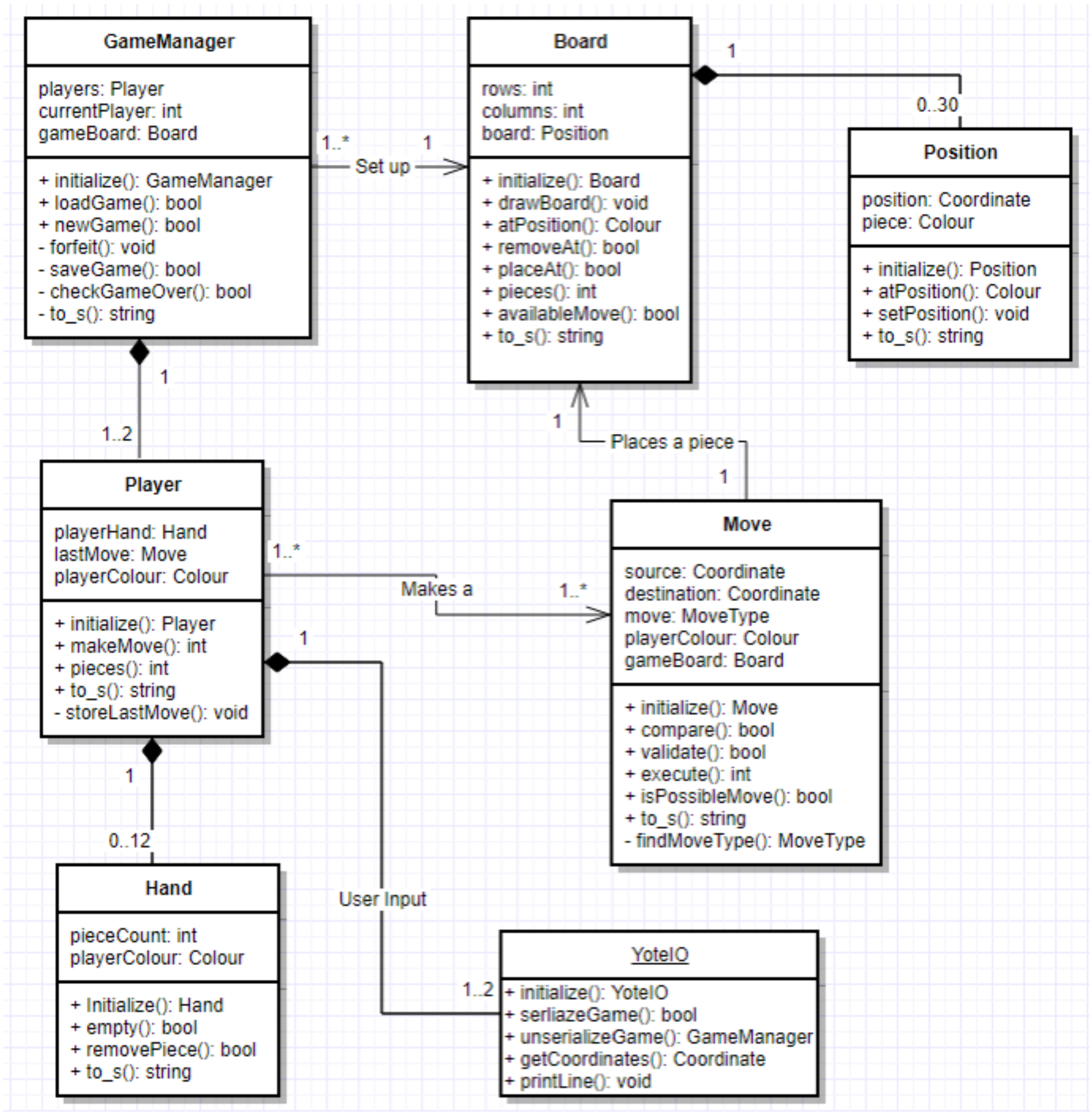

Figure 2 Class Diagram for Yoté

API of Classes

Table 1 API of Classes for Yoté

Class	Method Name	Arguments	Return Values	Description
GameManager	Initialize	Filename=nil	GameManager	Creates a new GameManager instance. If filename is not nil. Values are loaded from that file
	loadGame	Filename	Bool	Deserializes the file from filename and loads the values from that file into the current gamestate
	newGame	N/A	Bool	Begins a new game.
Board	Initialize	Int: rows int: columns	Board	Creates a new Board instance with size rows and columns.
	drawBoard	Player :white Player: black	Void	Draws the current board to the screen.
	atPosition	Coordinate: coord	Colour	Gets the colour of the piece at coord. :empty otherwise.
	removeAt	Coordinate: coord	Bool	Removes the piece at coord
	placeAt	Coordinate: coord, Colour: colour	Bool	Places the piece of colour ‘colour’ to the position at coord

	Pieces	Colour	Int	Gets the number of pieces of the colour ‘colour’ on the board
	availableMove	Colour	Boool	Checks if any pieces of ‘colour’ colour have any legal moves
	To_s		String	Serializes the board and the positions to a string
Player	Initialize	Colout: playerColour	Player	Creates a new player instance. The player will be the colour specified.
	makeMove	Board: gameboard	int	Used for the player to make their turn. Returns 0 if the move is successful, -1 for the player forfeiting and -2 for saving and quitting the game
	pieces	N/A	Int	Gets the number of pieces the player has in play
	To_s	N/A	String	Serializes the instance and returns the string

Move	Initialize	Coordinate: source, Coordinate: destination	Move	Creates a new Move instance. The instance variables source and destination will be initialized to the arguments.
	Compare	Move: lastMove	Bool	Checks if the current move is the same as the players last move, given as lastMove.
	Validate	N/A	Bool	Checks if the current move is legal or not
	findMoveType	N/A	Bool	Finds the type of move (Place, Move, Capture)
	Execute	N/A	Int	Executes the move. Returns 1 if the move is capturing the player and needs extra input from the player
	isPossibleMove	N/A	Bool	Checks if a piece can make any legal moves
	To_s	N/A	String	Serializes the instance and returns the string
YoteIO	Initialize	N/A	YoteIO	Creates a new YoteIO instance

	seralizeGame	GameManager: game, stirng: filename	Bool	Writes the current game state to file after serializing it.
	unserializeGame	String: filename	GameManger	Unserializes a previous game state and creates a new GameManager from those values
	getCoordiantes	String: prompt	Coordinate	Gets the coordinates of a move from the player
	printLine	String: print	N/A	Prints the string to stdout followed by a newline
Hand	initalize	Colour: playerColour	Hand	Creates a new Hand instance. The colour of all pieces in hand will be set to the argument.
	Empty	N/A	Bool	Checks if there are any pieces left in hand
	removePiece	N/A	Bool	Removes a piece from the hand. Used when placing a piece on the board.
	To_s	N/A	String	Serializes the instance and returns the string

Position	Initalize	Coordinate: position	Position	Creates a new position instance. The position instance variable is set to the argument.
	atPosition	N/A	Colour	Returns the colour of the piece
	setPosition	Colour: value	N/A	Set the colour of the instance variable position
	To_s	N/A	String	Serializes the instance and returns the string

Detailed List of Classes

Types

There are two main types that are used through this document that are not classes, but are important enough to make mention of and explain. Any time Coordinate or Colour is used in this document, it means these two types defined here.

1. **Coordinate** - This is the internal representation of the grid system for the game. It is just an array of 2 integers that hold the x, y coordinates. It is always the case that Coordinate[0] is the x coordinate, and Coordinate[1] is the y coordinate.
2. **Colour** - This is the representation of the colours/pieces of the game. It is the set of symbols { :white, :black, :empty }. Each player is assigned a Colour at the beginning of the game, and it can't be :empty. The purpose of :empty is for the Position class, as it holds the state of a single position on the board and needs a way to represent the absence of a piece.

GameManager

This class controls the main flow of the game. It contains the logic for starting and finishing a game. It also allows for the loading/saving of the game. This class should be instantiated once at the beginning of the program.

Members

- Player players[2] - The GameManager has 2 instances of Players, created on initialization of the object. The white player is in players[0] and the black player is in players[1].
- int currentPlayer - Variable that keeps track of which players turn it is, the possible values being [0, 1]. These numbers correspond to the index of the players[2] member, and should be used like this:
  players[currentPlayer].makeMove()
- Board gameBoard - Instance of the Board class, which keeps track of the game board. This is the only instance of the board class that should exist, and a reference to it is passed to the Player objects when it is their turn.

Public Functions

- GameManager initialize(filename=nil) - Constructor for the object. This takes an optional argument for a save file name. If file name is given, it will use the GameManager.loadGame() function to construct itself based on the save. If no file name is given, the constructor will set up the objects itself. Note that players[0] will be initialized with the :white symbol and the players[1] variable will be initialized with the :black symbol. That means that the currentPlayer variable will be initialized with the value of 0, meaning the white player will move first.
- bool loadGame(string: filename) - This function is used to load a serialized game file. The file name must be passed as a string, and it will destroy the current GameManager's Player and Board members and re-create them based on what has been saved. The

function will return true if the loading of the object succeeded, and will return false if any error has occurred. Possible errors include a missing file or incorrect type of file. It creates the GameManager object by calling the YoteIO.unserializeGame() function, then changing the current object's member variables to the ones from the object just loaded.

- bool newGame() - This function is used to start a new game. Once called, it will not return until the game is finished. It will have the main game loop, and will look like the following:

```
loop do
        ret = @players[@currentPlayer].makeMove(@gameBoard)

        # Need to check return and possibly call forfeit() or save(). See
        # Player.makeMove for return value, this part is not included.

        # Draw the board
        @gameBoard.drawBoard()

        ret = checkGameOver()

         if ret == false
        @currentPlayer = (@currentPlayer + 1) % 2
        else
          break
         end
end
```

# Print which player wins
Note the return of the function is true if the game was played through with no issues, and false if any error occurred.

Private Functions

- void forfeit(Colour: playerColour) - This function will end the game if one player has chosen to forfeit. It returns void, as it will end the program here with use of the exit! call built into ruby. It will print out which player has forfeited the game, and which player is the winner.
- bool saveGame() - This function is used to save and exit the game. It will use the default file name "YoteSave". The GameManager object will be saved by serializing itself. This is a process where it serializes itself, then serializes both Players and the Board. This function will simply print this object to a file, by use of the to_s this class implements. The output of to_s will be in YAML form, to allow for easy loading. Our aim is to implement the save/loading in the same way as done here: https://www.skorks.com/2010/04/serializing-and-deserializing-objects-with-ruby/.

- bool checkGameOver() - This function will check whether or not the game has finished. This function returns false if the game is not over, and returns true if the game is over. When checking if the game is over the following must be done:

1. Check if both players have 3 or less pieces using @players[i].pieceCount(). If both have 3 or less the game is over.
2. Check if the next player to move (ie. @players[(@currentPlayer + 1) % 2]) has an available move to make. This is done in a few ways. First, to check for an empty place that a piece can be placed into use Board.pieces(:empty), then you must make sure the Player has a piece they may place by using Player.pieces(). If both are not 0, then the player may place a piece. To check if the player has any pieces on the board they may move, use the Board.availableMove() function. This will return true or false based on if the Player has a move to make. If both of these conditions outlined are false, then the Player has no moves to make and the game is over.

- string to_s() - This function is used to serialize the GameManager object by printing out all important information in YAML form. Ruby has built in YAML support, so this should be straight forward. This function must also call to_s on both it's Player objects and it's Board object. It will also have to print the @currentPlayer variable.

Board

This class is used to represent the game board. It holds all information pertaining to the board, along with all the associated logic. There is only ever one instance of this class, and it is held by the GameManager object. Although the game only has a board size of 5x6, we will assume maybe this changes at some point so these values are not hard-coded, but rather passed into the object.

Members

- int rows - Variable that keeps track of how many rows the board has
- int columns - Variable that keeps track of how many columns the board has
- Position board[@rows][@columns] - This 2-D array of position objects is the board itself. This is created during the initialization of the object.

Public Functions

- Board initialize(int: rows, int: columns) - This function initializes a game board with the given dimensions. For the sake of this project, these should be 5 rows and 6 columns. This function then initializes a 2-D array of Positions in the @board array. Note that for @board[i][j], the i corresponds to the x-coordinate, and the j corresponds to the y-coordinate.
- void drawBoard(Player: white, Player: black) - This function is used to draw the board. For the sake of this assignment, that will mean this function uses the YoteIO class to write to stdout. This will have to contain all the logic for drawing the board, and the pieces in their proper spots. What characters will denote pieces will be implementer defined. The characters used to draw the board are also implementer defined, however

it is recommended to use the '|' character as the side walls, and the '-' character as the top walls. The main algorithm will include looping over each position of the board, using the atPosition() function to determine what is there, and then drawing it. See the YoteIO class for which functions may be used to write to stdout. This function must also print out the number of pieces each player has in their hand somewhere on the screen. This is why both Players are sent to this function, so the number of pieces may be determined by calling Player.pieces(). Note this will return the total number of pieces in play for the Player. To determine how many pieces the Player has in their hand, subtract off the result of Board.pieces() call. The board should also be printed with markings for each square in a manner similar to chess (along the sides). The x-axis should be marked with the numbers 0-5, while the y-axis should be marked with the letters 'a'-'e'. This allows the user to reference squares easily.

- Colour atPosition(Coordinate: coord) - This function is used to determine what is at a given position of the board. It will return the Colour type, which as defined above is one of {:white, :black, :empty}. It takes a Coordinate as an argument which is defined above as [int x, int y]. Should an incorrect coordinate be given, or one that does not exist on the board, :empty will be returned. This function uses the Position object's function with the following parameters: @board[coord[0]][coord[1]].atPosition().
- bool removeAt(Coordinate: coord) - This function is used to remove a piece that is at a given position. The function takes the Coordinate to remove as it's argument. It uses the Position object's function to do this: @board[coord[0]][coord[1]].setPosition(:empty). It will return true if the piece was successfully removed, and will return false if the piece could not be removed. The following will cause a failure to remove (and a return of false) and should always be checked:

1. Invalid Coordinate, one that is off the Board
2. The Coordinate is empty

- bool placeAt(Coordinate: coord, Colour: colour) - This function is used to place a piece of a given Colour at the Coordinate given. It takes both the Coordinate and Colour as arguments. It uses the Position object's function to do this: @board[coord[0]][coord[1]].setPosition(colour). It will return true if the piece is successfully added, and false if it could not be added. The reasons it could not be added are (must be checked in this function):

1. The Coordinate given is not empty
2. The given Coordinate is invalid (outside scope of board)
3. The Colour given is :empty. This would be functionally the same as using removeAt(), and thus that function should be used instead.

- int pieces(Colour) - This function returns the number of pieces on the board for a given Colour. It is used by GameBoard to check if the game has ended or not. This function just loops over the @board[@rows][@columns] member and counts how many of the given Colour exist.
- bool availableMove(Colour) - This function is used to check if a given Colour has any available moves on the board. This includes both moving a piece and capturing a piece. The function works by looping through the @board array and finding all pieces that are of type Colour. Then, it creates a Move object with that source coordinate, and uses the

Move.possibleMoveExists() to check if that piece has a valid move. If it does, then immediately return true. If there is no piece for the given Colour that the Move.possibleMoveExists() returns true for, then the Player has no available move, so return false.

- string to_s() - This function is used to serialize the Board object. It must print all it's member variables and then calls the to_s() method on all of the Position objects.

Player

The Player class defines a single user of the Yote game. There will be two instances of the Player class held by the GameManager class. These two instances will never directly interact. Each one defines the important attributes and logic for the player.

Members

- Hand playerHand - Variable that will contain the player's hand. This is where the player's pieces are stored.
- Move lastMove - This variable holds the last move the player made. This is to ensure the player's move does not violate the rules by backtracking a piece.
- Colour playerColour - Variable that holds what colour the player is, so that when making a move, the Move class knows which player made the move.

Public Functions

- Player initialize(Colour: playerColour) - This function initializes a Player. It takes the colour of the player (either :white or :black) as the argument. Then the @playerHand is initialized using the default Hand constructor, and the @lastMove is initialized with the value nil.
- int makeMove(Board: gameboard) - This function is used to make a Player's turn occur. It is called by GameBoard at the beginning of the player's turn. It accepts a reference to the current Board, to allow for it to make moves. This function returns the following: 0 = Successful move made; -1 = Player wants to surrender; -2 = Player wants to save and quit game. This function has a decent amount of logic outlined below:

1. Get the user's "source" coordinate via the YoteIO class. Note that if this coordinate is entered blank, then the player has selected to place a piece instead of move a piece. This must be relayed to the user in the call to YoteIO in a way such as YoteIO.getCoordinates("Please enter source coordinate (Leave blank to place piece"). See YoteIO for details of this function call.
2. Get the user's "destination" coordinate via YoteIO class in same manner as step 1.
3. Check that the coordinates are not a forfeit ([-1, -1]) or a save and quit ([-2, -2]). These values are what YoteIO.getCoordinates() returns when the user has requested either, refer to that function for more details. If the coordinates are a forfeit or save and quit, return this function immediately with the value -1 or -2 respectively.
4. Create the Move object with these two coordinates, and then immediately call Move.validate() to ensure the move is valid. If it is not, then start at step 1 again. That

means step 1-4 must be in a loop until valid a valid move, forfeit, or save and quit is entered.

5. Once the move is validated, check if the source coordinate entered is nil, and if it is use the @playerHand to ensure the Player has a piece to place. If the Player does not have a piece to place (Hand.getPiece() returns false) then use YoteIO to relay this message to the user. Then, begin at step 1 again.
6. If the source coordinate was not nil then this is a move or capture. It must be checked against the Player's previous move to ensure the Player is not back-tracking. To do this, use the Move object created in step 4 to call the method Move.compareLastMove(). See this function in the Move class for more details. Again if this check fails, begin at step 1 again.
7. Once all these checks have been done and passed, call Move.execute(). If the return value of execute is 0, then go to step 9. If it is 1, proceed to the next step.
8. If Move.execute() returned 1, then there was a successful capture and the User must be prompted to get which piece they want to remove from the board. Use the YoteIO class to get Coordinates from the User, with a prompt such as "Select piece to be removed". Next, validate there is a piece to be removed at that position with Board.atPosition() (ensuring proper Colour being checked). If there is a piece there to remove, use Board.removeAt() to remove it. Keep doing this until the user enters a valid piece. Note that if execute returns 1 there **must** be a piece to remove somewhere on the Board, so don't feel bad about forcing the user to enter a valid coordinate.
9. Use the storeLastMove() to store this executed move in the Player's member variable @lastMove. Lastly, return 0.

- int pieces() - This function returns how many pieces the player has **in play**. The in play part is a very important distinction. This means amount of pieces both in the Hand, and on the Board. The only pieces not counted are ones that have been captured by the opponent. This means this function must call Hand.count(), and then add that value to Board.count(Player.Colour). This will return all the pieces in both the hand and in play. The purpose of the function is to be used by GameManager to get the amount of pieces both players have to ensure it is greater than 3 and the game may continue.
- string to_s() - This function is used to serialize the object. It must print out all member variables, and call the to_s() method on it's Hand and lastMove instance variables.

Private Functions

- void storeLastMove(Move: lastmove) - This function is used to store the move the Player just made. This allows the Player to compare next turn's move to ensure it is valid. All this function does is stores the lastmove variable passed into the function in the @lastMove instance variable. Since this function should not fail, it will return nothing.

Move

The Move class defines all functionality around making a move during the game. This includes finding if a piece has valid moves, validating a given move, and executing the move once valid.

This class is used by Player when creating a move, but is also used by Board when determining if a Player still has a valid move to make. This class makes heavy use of the Coordinate type, which as noted above is an array of 2 integers that are the x,y coordinates of the piece.
The difference between source and destination Coordinates are important. Both are **always** created upon instantiation of the object, no matter if it is a placement or movement. When the Move is a placement, the source Coordinate will be nil, while the destination Coordinate will be the location on the Board to place the piece. When the Move is a capture or move, the source Coordinate will be the location of the piece that is being moved, and the destination will be the location the piece will end up in.
The MoveType type will be an internal representation of the type of Move that is occurring. This simplifies some of the logic in the main functions. This will be set during the initialize() function, and will have the values { :move, :capture, :placement, :illegal }.

Members

- Coordinate source - Variable that holds the source location of the piece. If the move is a placement of the piece, this Coordinate will be nil.
- Coordinate destination - Variable that holds the destination of the move. This variable should **never** be nil, it must always have a location except when using the Move object for Move.possibleMoveExists(), then the value of this variable does not matter.
- MoveType move - Variable that holds the type of Move that is occurring.
- Colour playerColour - This variable holds the Colour of the Player that is moving. This is needed for validating the move that is taking place, so that errors such as capturing your own pieces do not occur.
- Board gameBoard - Reference to the current Board state, set on initialization.

Public Functions

- Move initialize(Coordinate: source, Coordinate: destination, Colour: playerColour, Board: gameBoard) - This function is used to initialize the object with the given values. To determine the type of Move, the private function findMoveType() should be used.
- bool compare(Move: lastMove) - This function is used to compare whether the move trying to be made is a back-tracking move. It is illegal to move a piece back into the location it was moved out of in the previous turn. The function returns false if this move is legal, and true if the move is illegal. This function should follow these steps:

1. If lastMove.moveType() is equal to :illegal, :placement, or :capture then immediately return false.
2. If lastMove.destination != @source then return false, as they are not moving the same piece they moved last turn.
3. If lastMove.source == @destination then this is an illegal move, because the Player is attempting to move back into the location they were the previous turn.

- bool validate() - This function is used to validate whether the created Move is legal or not. It is actually just a wrapper for the @moveType member, as the legwork is done upon initialization of the object by the findMoveType() function. This function should return false if @moveType == :illegal and true otherwise.

- int execute() - This function is used to execute the Move that has been created. The function uses the @gameBoard member to remove and add the pieces where needed. There are a few different steps based on what type of move it is:

1. If @move == :illegal return -1 immediately
2. Always place a piece at the destination Coordinate using @gameBoard.placeAt(@destination)
3. If @move == :placement return 0 as the function is finished.
4. If @move == :movement then the piece must be removed from where it used to be by doing @gameBoard.removeAt(@source), then return 0 as function is finished.
5. If @move == :capture then remove the piece at the source the same way as step 4, then remove the piece that was jumped over by calculating the middle piece (See findMoveType() step 4 for how to do this).
6. The player must then remove another piece from the board if there is one to remove. First, check if there is a piece to remove by calling Board.pieces() with the opposite colour of the player moving. If this is 0, then return 0 as the function is finished. If there is a piece that may be removed, Player must be alerted to this as they have to prompt the User for which piece to remove. If this is the case, return 1 to signify to Player they need to get another piece to remove.

- bool isPossibleMove() - This function is used by GameManager to determine if a given Player still has moves. This will check the given source coordinate of the Move to see if there is any open moves for that piece to make. It returns true if there is a valid move for that piece, and false if there is not. The steps for this function are:

1. Check the 4 adjacent coordinates to see if any are open using @gameBoard.atPosition(). If one of these Coordinates are empty (atPosition returns :empty), then return true as a valid move exists.
2. If a piece of the opposite Colour sits in one of the 4 adjacent positions, then check the next position after that to see if it is empty. If that Coordinate is empty, then there is a valid capture move so return true.
3. If none of these options are possible, return false.

- string to_s() - This function is used to serialize the object. It must print out all the member variables of the object.

Private Functions

- MoveType findMoveType() - This function determines what type of move is taking place based on the source and destination coordinates.
- If source is nil, then @move=:placement if the destination Coordinate is empty. This may be checked by using @gameBoard.atPosition(@destination) == :empty. If the Coordinate is not empty, then @move=:illegal.

1. If either of the Coordinates are out of bounds, then @move=:illegal, and the validation function will automatically return false. So checks should be done that the pieces are not out of bounds by using the Board instance passed into this function.
2. To find out if it is a movement of a piece, check if either the x Coordinate **or** the y Coordinate of the destination differ by exactly 1 from the source coordinate. If this is the case then set @move=:move. Note that if **both** x and y differ by 1 then the move is illegal as it would be a diagonal movement. If this is the case, set @move=:illegal. It must also be checked that @gameBoard.atPosition(@source) ==   @playerColour, and @gameBoard.atPosition(@destination) == :empty. If either of these are not true, then set @move=:illegal.
3. To find out if it is a capture, check that either the x or y Coordinate of the destination is exactly 2 units from the source Coordinate. If x is 2 away, then y **must** be the same and vice versa. If this is the case, then 3 checks must happen, first check if the source coordinate has the right piece @gameBoard.atPosition(@source) == :@playerColour, then check the destination is empty @gameBoard.atPosition(@destination) == :empty. Lastly, it must be determined if a piece is being "jumped over". This can be done by finding the Coordinate being "jumped over", and then checking it has the opposite colour. This following snippet illustrates an easy way to do this check:

```
x = 0, y = 1 # For easier reading
jumped = [(@destination[x] + @source[x]) / 2, (@destination[y] + @source[y]) / 2]

result = (@gameBoard.atPosition(@jumped) != :empty &&
                    @gameBoard.atPosition(@jumped) != :playerColour)

if result == true
@move = :capture
else
 @move = :illegal
end
```

YoteIO

YoteIO is the class that controls the whole programs IO. It is meant to decouple the other classes from user input, and create a uniform interface for communicating with the user. The name is due to Ruby already having an IO class, so trying to also have an IO class breaks everything. Note there are no members or private functions within this class.

Public Functions

- YoteIO initialize() - This function initializes the YoteIO class, it has no behaviour of note.
- bool serializeGame(GameManager: game, string: filename) - This function is used to serialize the GameManager object. It does this by using YAML::dump(game) on the GameManger, then printing the result of this function to the file given by filename. Return true unless the dump function causes an error, then return false.

- GameManager unserializeGame(string: filename)- This function loads the serialized object saved in the file with the name filename by using the function YAML::load(). Again, the method to serialize/unserialize may be located at the URL in the GameManager.loadGame() description.
- Coordinate getCoordinates(string: prompt) - This function is used to get the coordinates from the user. It will print out the prompt using YoteIO.printLine(), then receive input from the user. There are 4 distinct cases of user input this function may handle:

1. The user enters the string 'forfeit'. In this case, return the Coordinate [-1, -1]. This tells the caller of this function the Player wants to forfeit.
2. The user enters the string 'save'. In this case, return the Coordinate [-2,-2].
3. The user enters the encoded square in the form 'a1' or '1a'. As noted above, the squares are marked along the x-axis with numbers, and along the y-axis with letters. This function will parse the input coordinates into the proper form. That is, convert the letter in the string into it's corresponding number (with 'a' = 0). Then place into the Coordinate with the x-coordinate first, then the y-coordinate. This coordinate should then be returned.
4. The user enters input that does not match the above cases. When this occurs, inform the user using @YoteIO.printLine() that their input is invalid. Then, re-print the prompt and try again. This function should never return until proper input is given from the user.

- void printLine(string: print) - This function prints the given string to stdout. It will also print the newline character.

Hand

The Hand class represents the pieces that the Player has that are not yet placed on the Board. It is used to ensure the Player does not place more pieces than they have.

Members

- int pieceCount - This member variable represents how many pieces the player has in their hand.
- Colour playerColour - Colour of the Player that this hand is used by

Public Functions

- Hand initialize(Colour: playerColour) - The constructor for Hand. This takes the Colour of the Player as an argument. Note that @pieceCount must be set to 12 when initialized.
- bool empty() - This function is used to determine if the Hand is empty. Returns true if @pieceCount == 0 and false otherwise.
- bool removePiece() - This function is used to remove a piece from the Hand. It is used by Player to to determine if they have a piece that may be removed from the Hand, then removes one from the Hand. If there is no piece to be removed (checked with a call to Hand.empty), then this function returns false. If there is a piece to be removed, then decrement @pieceCount by 1 and return true.

- string to_s() - This function is used to serialize the object. It must print out all the Hand member variables.

Position

The Position class defines a single square on the board. It is used to store what is at the square at any given time. It is used by the Board to keep track.

Members

- Coordinate position - This variable defines where the position is on the Board.
- Colour piece - This variable holds what is at the square. For possible values see Colour under the Types section.

Public Functions

- Position initialize(Coordinate: position) - This function initializes the Position object and sets the @position member to the argument position. Then, the @piece member is set to :empty.
- Colour atPosition() - This function is used to get the value of @piece.
- void setPosition(Colour value) - This function is used to set what the Position holds. It will set the value of @piece to the value of the argument. This function returns no value.
- string to_s() - This function is used to serialize the object. It must print out all the Position member variables.

Sequence Diagrams

Starting A Game

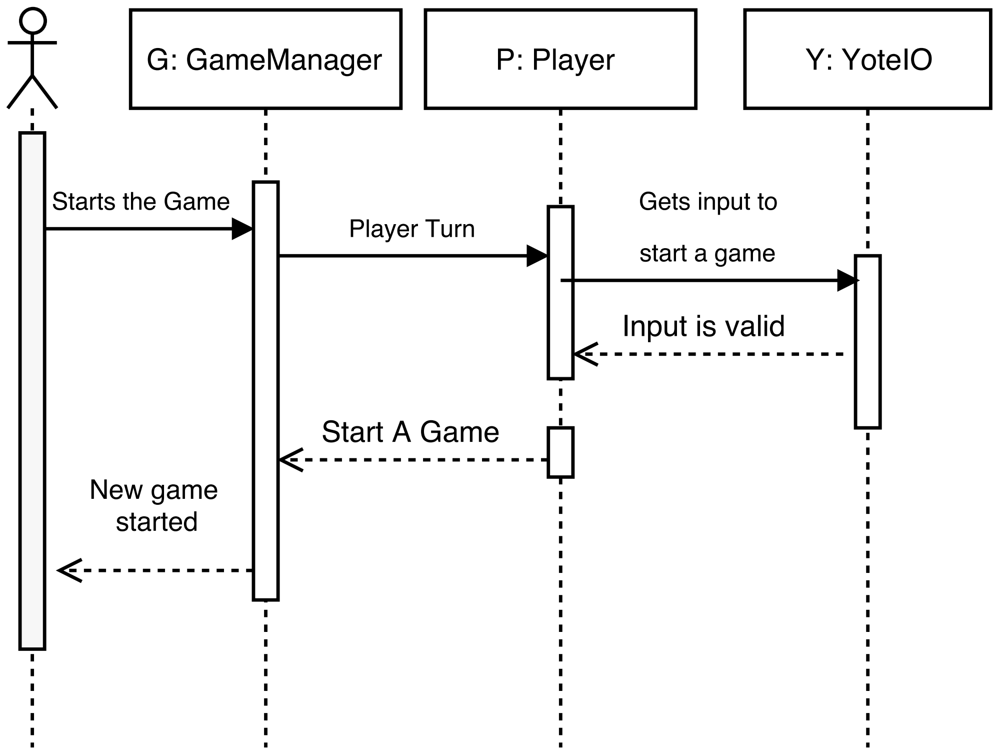

Figure 3 Sequence Diagram for Starting a Game

Forfeit a Game

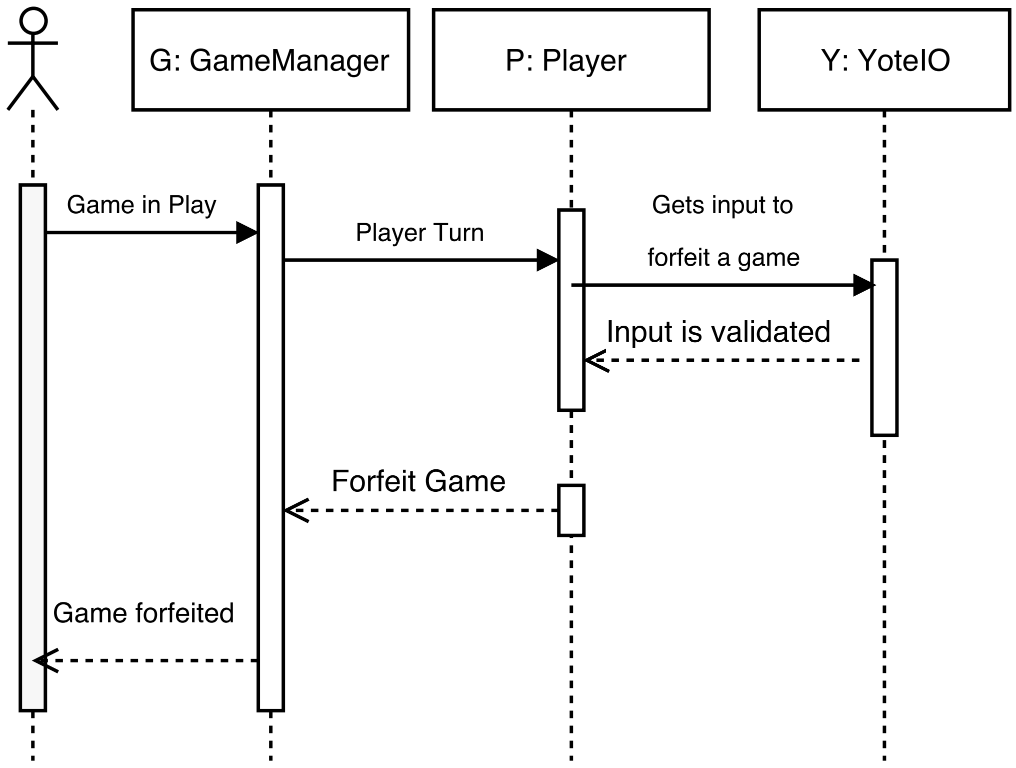

Figure 4 Sequence Diagram for Forfeiting a Game

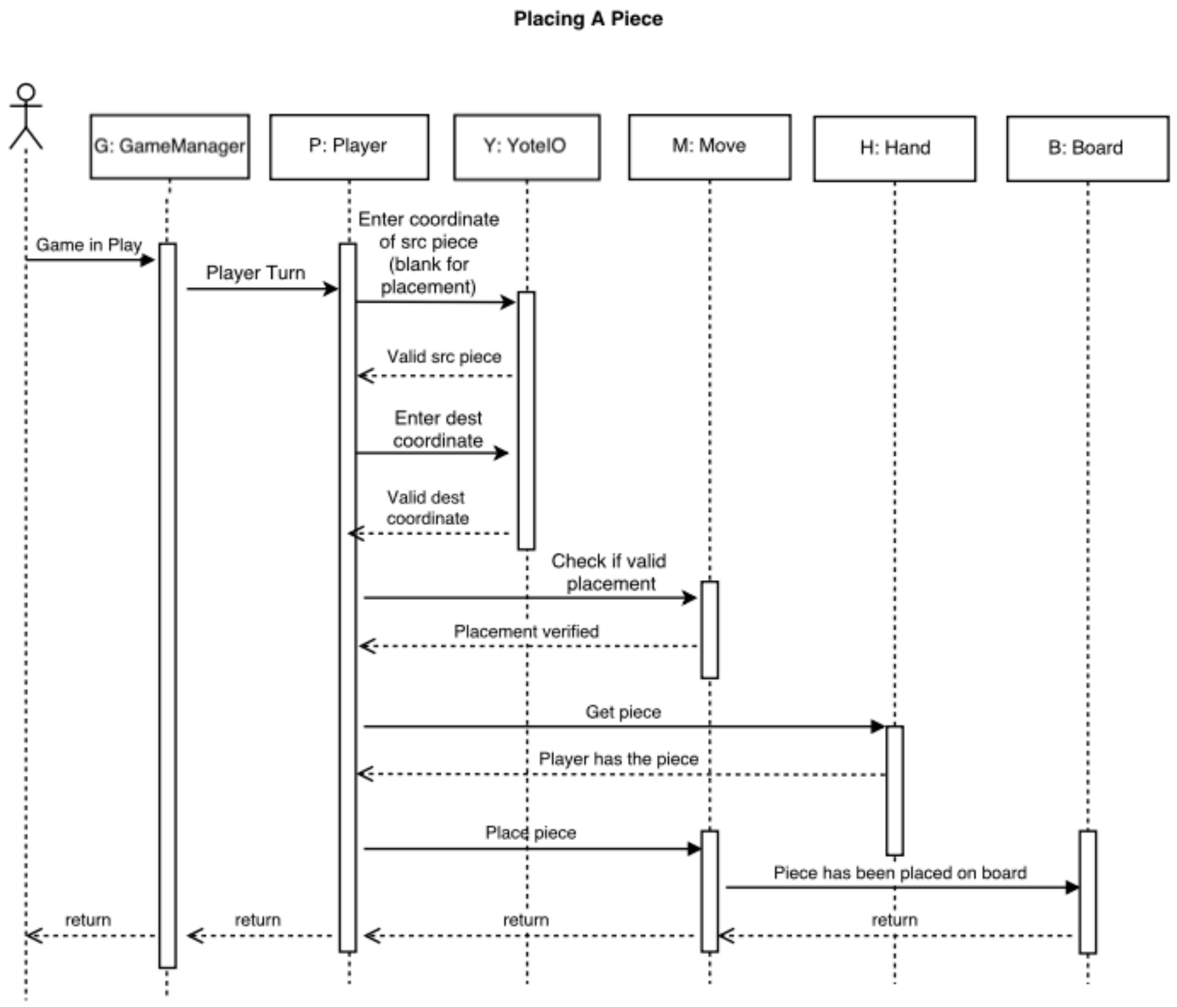

Figure 5 Sequence Diagram for Placing a Piece

Moving a Piece

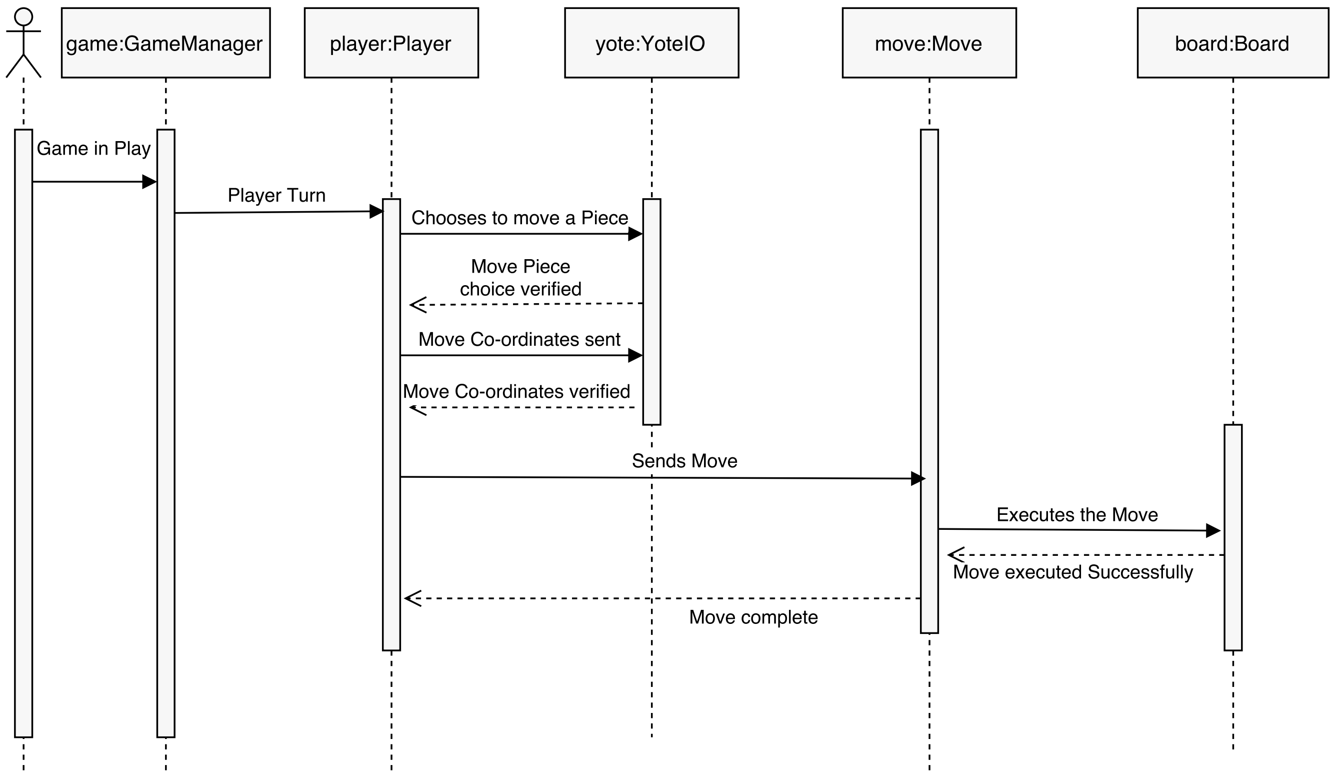

Figure 6Sequence Diagram for Moving a Piece

Capturing a Piece

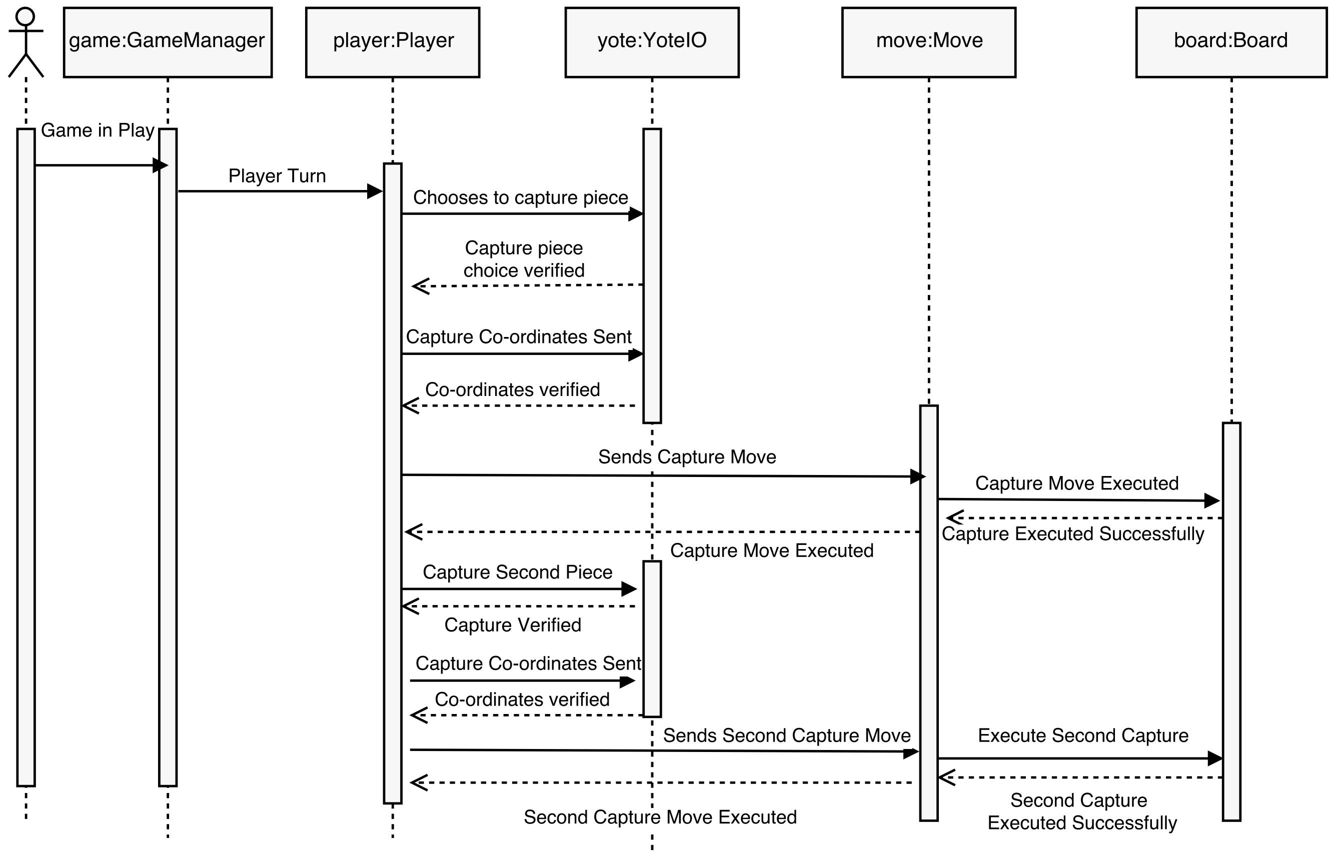

Figure 7 Sequence Diagram for Capturing a Piece

Saving A Game

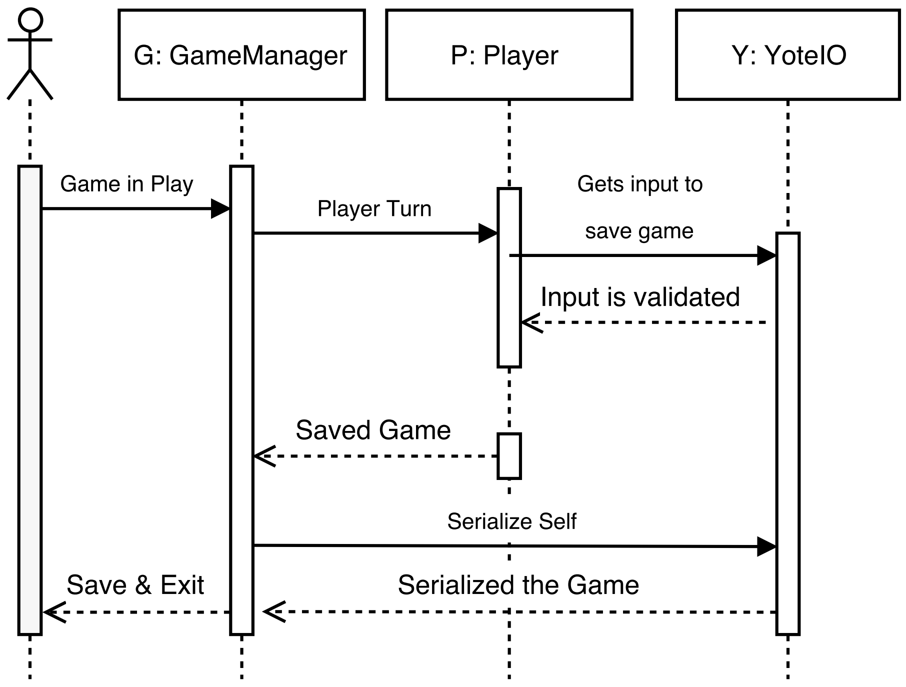

Figure 8 Sequence Diagram for Saving a Game

Use Cases

UC1: Starting a new game

Stakeholders List:

- ❖ Player One who wants to play a new game.
- ❖ Player Two who wants to play a new game.

Primary Actor: Player One.

Goal: To begin a new game of Yote.

Initiating Event: Player One and Player Two want to play a new game.

Basic Flow (Main Success Scenario):

1. Player clicks start.
2. Player One finds another player so they can play a new game.
3. Player One requests the system to make a new game.
4. The system creates a new game for the players.
5. The two players begin to play the game.

Extensions (Alternate Scenario):

2.a: Player One requests the system to load a previously saved game.

2.a.1: Player selects the previously saved game.

2.a.2: The system loads the previously saved game for the players.

2.a.3: The players begin to play the loaded game.

UC2: Forfeiting a game

Stakeholders List:

- ❖ The player who wants to forfeit the game.
- ❖ The opposing player who is playing against the player.

Primary Actor: The forfeiting player.

Goal: To end the game early because they no longer want to play.

Initiating Event: Upon turn, the player is finding the current game challenging so they decide they want to forfeit the game.

Basic Flow (Main Success Scenario):

1. Player decides they want to forfeit the game.
2. Player informs the system that they are forfeiting.
3. The system requests confirmation for the forfeit.
4. The systems informs the opponent that the player has forfeited.
5. The system ends the game and the player’s opponent is declared the winner.

Extensions (Alternate Scenario):

3a.: Player decides that they do not wish to forfeit the game.

3a.1. Player selects cancel.

3a.2. The system does not end the game and the game continues.

UC3: Placing a piece on the board

Stakeholders List:

- ❖ The player waiting to make a move.
- ❖ The opponent player impacted by the move.

Primary Actor: The player who is placing a piece on the board.

Goal: The player wants to place one of their new pieces on the board.

Initiating Event: Upon turn, the player decides to place a new piece on the board.

Basic Flow (Main Success Scenario):

1. Player takes a piece from their hand.
2. Player chooses a position to place a piece on the board.
3. Player places the piece on the board.

Extensions (Alternate Scenario):

1a: Player has no pieces in hand.

1a.1. Player cannot place a piece on the board.

1a.2. Player must select another action for their turn.

2a.: Player selects an invalid position.

2a.1. Player tries to place a piece on already occupied position.

2a.2. Player chooses a different position on the board until it is valid.

UC4: Moving a piece on the board

Stakeholders List:

- ❖ The player waiting to make a move.
- ❖ The opponent player impacted by the move.

Primary Actor: The player who is moving an existing piece on the board.

Goal: Move an existing piece to a new position on the board.

Initiating Event: Upon turn, the player decides to move an existing piece on the board.

Basic Flow (Main Success Scenario):

1. Player selects an existing piece to move on the board.
2. Player moves their existing piece in a valid direction (vertically or horizontally) to a new position on the board.

Extensions (Alternate Scenario):

2a. Player moves the piece to an invalid position:

2a.1. Player selects another position to move their piece which is going to a new position they were not previously occupying (vertically or horizontally).

2b. The player is not able to move anywhere due to no more positions available:

2b.1. The game will come to an end.

2b.2. Player with most captures wins the game.

UC5: Capturing a piece on the board
Stakeholders List:

- ❖ Player wants to capture a piece from game board.
- ❖ Opponent Player wants to verify if player move will lead to capturing a piece from game board.

Primary Actor: Player whose turn it is.
Goal: Upon turn, player wants to capture opponent players piece.
Initiating Event: Player moves their piece and wants to capture their opponent's piece.
Basic Flow (Main Success Scenario):

1. Upon turn, the player makes their move on game board.
2. As a result of players move, the player is now able to capture an opponent's piece.
3. Opponent Player verifies that their piece is orthogonally adjacent to the player piece.
4. Player captures their opponents orthogonally adjacent piece.
5. Player gets another turn to capture another one of opponent's piece.
6. Player looks at all opponent's pieces on game board.
7. Player strategically captures another one of opponent's pieces.

Post-Condition: The number of pieces captured by player increases by 1.
Extensions (Alternate Scenario):
3a. Opponent's piece is not orthogonally adjacent to players piece:
3a.1. Opponent Player notifies player about their piece.
3a.2. Player does not capture their opponent's piece.
3a.3. Opponent Player plays their next move.
5a. All of the opponent player's pieces on the board have been captured.
5a.1. Game will come to an end.
5a.2. Player with most captures wins the game.

UC6: Player wants to save the game for a later time
Stakeholders List:

- ❖ Player that wants to save a game.

Primary Actor: Player who wants to save the game
Goal: To save the game so the player is able to come back to it at a later time.
Initiating Event: When the player clicks save.
Basic Flow (Main Success Scenario):

1. Player starts a new game.
2. Player plays a round of the game.
3. Player clicks save.
4. Both players now have the opportunity to leave the game and come back to load it at a later time, rather than starting another new game.

Extensions (Alternate Scenario): N/A
Use Cases Utilize: N/A

Entity Diagram

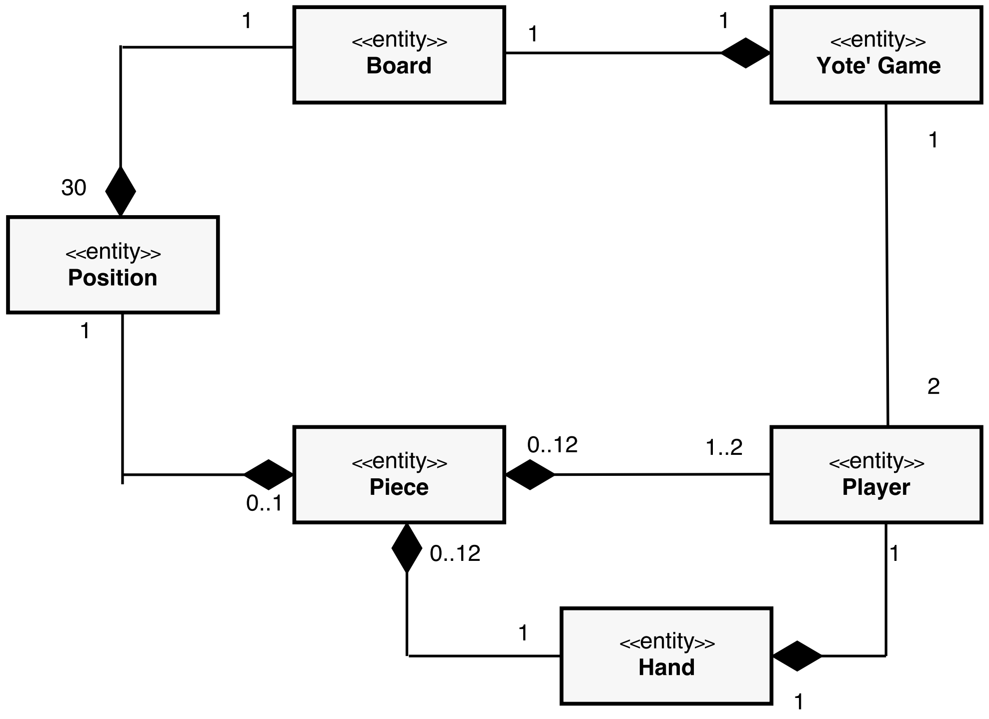

Figure 9 Entity Diagram for Yoté Game

Appendix A: CRC Cards

Table 2 Use Case: Starting a new game

Class: GameManager Ask currentPlayer to begin or load a game Begin a new game	Collaborators: Player Instance Variables: currentPlayer Methods: checkGameOver(), newGame()
Class: Player Create input to begin a game Give input to YoteIO Give input to GameManager	Collaborators: GameManager, YoteIO Instance Variables: Methods: makeMove()
Class: YoteIO Validate input Return input to Player	Collaborators: Player Instance Variables: Methods: None

Table 3 Use Case: Forfeiting a game

Class: GameManager Ask Player for their move End the game	Collaborators: Player Instance Variables: currentPlayer() Methods: endgame()
Class: Player Creates input to forfeit the game Give input to YoteIO Give input to GameManager	Collaborators: GameManager, YoteIO Instance Variables: Methods: makeMove()
Class: YoteIO Validate input Return input to Player	Collaborators: Player Instance Variables: Methods: None

Table 4 Use case: Place a piece on the board

Class: GameManager Tell Player to make a move	Collaborators: Player Instance Variables: currentPlayer Methods:
Class: Player Give input to YoteIO for validation Give move to 'Move' for validation Request a piece from Hand Give the piece to Move	Collaborators: GameManager, YoteIO, Move, Hand, Board Instance Variables: lastMove Methods: makeMove()
Class: YoteIO Validate input from player	Collaborators: Player Instance Variables: Methods:
Class: Move Validate the move Give the piece to Board	Collaborators: Player, Board Instance Variables: source, destination, move Methods: compare(), validate(), findMoveType(), execute()
Class: Hand Return a piece to the player	Collaborators: Player Instance Variables: numPieces [] Methods: getPiece()
Class: Board Place the piece on the board	Collaborators: Move Instance Variables: rows, columns, board [] [] Methods: placeAt()

Table 5 Use case: Moving a piece on the board

Class: GameManager Tell player to make a move	Collaborators: Player Instance Variables: currentPlayer Methods:
Class: Player Give input to YoteIO for validation Give to move to 'Move' for validation Give move to 'Move' for execution	Collaborators: YoteIO, Move Instance Variables: lastMove Methods: makeMove()
Class: YoteIO Validate input from player	Collaborators: Player Instance Variables: Methods:
Class: Move Validate move from player Send the movement positions to board	Collaborators: Player, Board Instance Variables: source, destination, move Methods: compare(), validate(), findMoveType(), execute()
Class: Board Move the piece from the source position to the destination position as specified in the move	Collaborators: Move Instance Variables: rows, columns, boards [] [] Methods: removeAt(), placeAt()

Table 6 Use case: Capturing a piece

Class: GameManager Tell player to make a move	Collaborators: Player Instance Variables: currentPlayer Methods:
Class: Player Give input to YoteIO for validation Give to move to 'Move' for validation Give input to YoteIO for validation Give capture selection to 'Move' for validation Give move to 'Move' for execution	Collaborators: YoteIO, Move Instance Variables: lastMove Methods: makeMove()
Class: YoteIO Validate input from player	Collaborators: Player Instance Variables: Methods:
Class: Move Validate move from player Give source and destination to board to see if they are occupied or not Ask player to select another piece to capture Validate capture selection from player Send the movement positions to board Send the capture positions to board	Collaborators: Player, Board Instance Variables: source, destination, move Methods: compare(), validate(), findMoveType(), execute()
Class: Board Check if the source and destination positions are empty Move the piece from the source position to the destination position as specified in the move Remove the pieces specified in the capture selection	Collaborators: Move Instance Variables: rows, columns, boards [] [] Methods: removeAt(), placeAt(), atPostion()

Table 7 Use case: Saving a game

Class: GameManager Tell player to make a move Tell YoteIO to serialize the gamestate	Collaborators: Player, YoteIO Instance Variables: currentPlayer Methods: saveGame()
Class: Player Creates input to save the current game Give input to YoteIO for validation Give input to GameManager	Collaborators: GameManager, YoteIO Instance Variables: Methods:
Class: YoteIO Validate input from player Serialize the GameManager object to a file on the disk	Collaborators: Player, GameManager Instance Variables: Methods: seralizeGame()