# **Design Specification**

A Tic Tac Toe game where the AI can beat players when they mess up at most one move.

This specification cover: Class Diagram, Glossary of the system, CRC Cards, State Diagram, Sequence Diagram, Overall Test Plan and Rationale.

**BY**

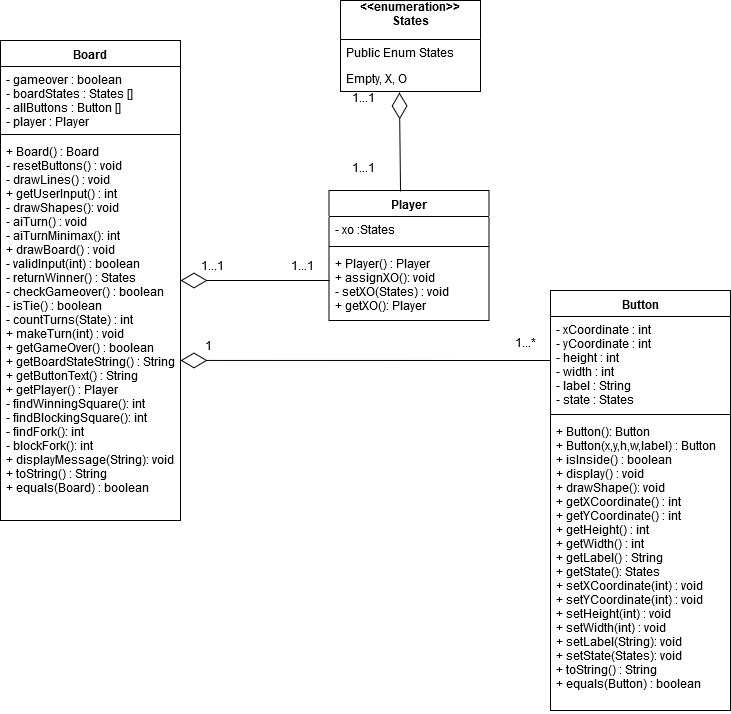
***Sean Masterson - 018653932***

***Alex Banh - 015428502***

***Brandon Walker - 015726800***

***Hieu Pham - 025450189***

**Class Diagram**



**Glossary of the System**

* Board: This class represents a 3x3 tic-tac-board game.
  + Variables:
    - boolean gameover
      * Stores whether the game is over.
    - Button[] allButtons
      * The 9 buttons which the player can press to make a move
    - Player player
      * The human player
  + Methods:
    - Board() - Board
      * Default board constructor
    - resetBoard() - void
      * Clears the board and resets it
    - initializeButtons() - void
      * Initialize all the buttons on the board to be used
    - drawLines() - void
      * Draws the lines separating the buttons on the board
    - getUserInput() - int
      * Read in user input and turns it into an edit on one of the buttons on the board
    - drawShapes() - void
      * Calls the drawShape() function of each button
    - aiTurn() - void
      * Makes a move for the opposing player
    - aiTurnMinimax() - int
      * Returns 1 if the current board space will result in a win, 0 if it will result in a tie, -1 if it will result in a loss
    - drawBoard() - void
      * Draws the current game state of the Tic-Tac-Toe game by calling drawLines() and drawShapes()
    - validInput(int) - boolean
      * Checks to see if the user made a valid choice for the program
    - returnWinner() - States
      * Return whether X won, O won or if its a tie
    - checkGameover() - boolean
      * Check to see if X or O won or if it tied
    - isTie() - boolean
      * Checks the board to see if the player tied with the AI
    - countTurns(State) - int
      * Count how many turns the Player and AI has moved
    - makeTurn(int) - void
      * Allows the Player and AI to make a move as long as the game is not over
    - getGameOver() - boolean
      * Returns whether or not the game is finished or not
    - getAllButtonsString() - String
      * Returns all of the buttons’ states as a string
    - getButtonText() - String
      * Gets the button’s to string method and prints it out for the array
    - getPlayer() - Player
      * Return a player
    - findWinSquareForPlayer() - int
      * Returns index of an open square that will cause the player to win, or -1 if does not exist.
    - findWinSquareForAI() - int
      * Returns index of an open square that will cause the AI to win, or -1 if does not exist.
    - findFork() - boolean
      * Returns true if the player can make a fork, false otherwise
    - blockFork() - boolean
      * Returns true if the player can block a fork from the AI, false otherwise
    - displayMessage(String message) - void
      * This function will display the message passed in as a String.
    - toString() - String
      * Return the board in a text based way
    - equals(Board) - boolean
      * Check to see if two boards are equal
* Button: This class represents a clickable button
  + Variables:
    - int xCoordinate
      * The x coordinate of the button
    - int yCoordinate
      * The y coordinate of the button
    - int height
      * The height of the button
    - int width
      * The width of the button
    - String label
      * The label of the button
  + Methods:
    - Button() - Button
      * Default button constructor
    - Button(xCoordinate,yCoordinate,height,width,label) - Button
      * Overloaded button constructor
    - isInside() - boolean
      * Check if the mouse cursor is within the button’s space on the screen
    - drawShape() - void
      * Draws the button's state.
    - display() - void
      * Displays the button with the coordinates and size as determined by member variables.
    - getXCoordinate() - int
      * Return button’s x coordinate
    - getYCoordinate() - int
      * Return button’s y coordinate
    - getHeight() - int
      * Return button’s height
    - getWidth() - int
      * Return button’s width
    - getLabel() - String
      * Return button name
    - getState() - States
      * Returns button’s state
    - setXCoordinate(int) - void
      * Set the button’s x coordinate
    - setYCoordinate(int) - void
      * Set the button’s y coordinate
    - setHeight(int) - void
      * Set the button’s height
    - setWidth(int) - void
      * Set the button’s width
    - setLabel(String) - void
      * Set the button’s label
    - setState() - void
      * Sets button’s state
    - toString() - String
      * Prints out the Button object as a string
    - equals() - boolean
      * Compares if two buttons are equal
* Player: This class represents a human player.
  + Variables:
    - States xo
      * The assigned state the player was assigned (X or O)
  + Methods:
    - Player() - Player
      * Creates an empty player object
    - assignXO() - void
      * Give a player an X or an O
    - setXO(States) - void
      * Sets a player in a chosen State
    - getXO() - Player
      * Return what state the player is in

**CRC Cards**

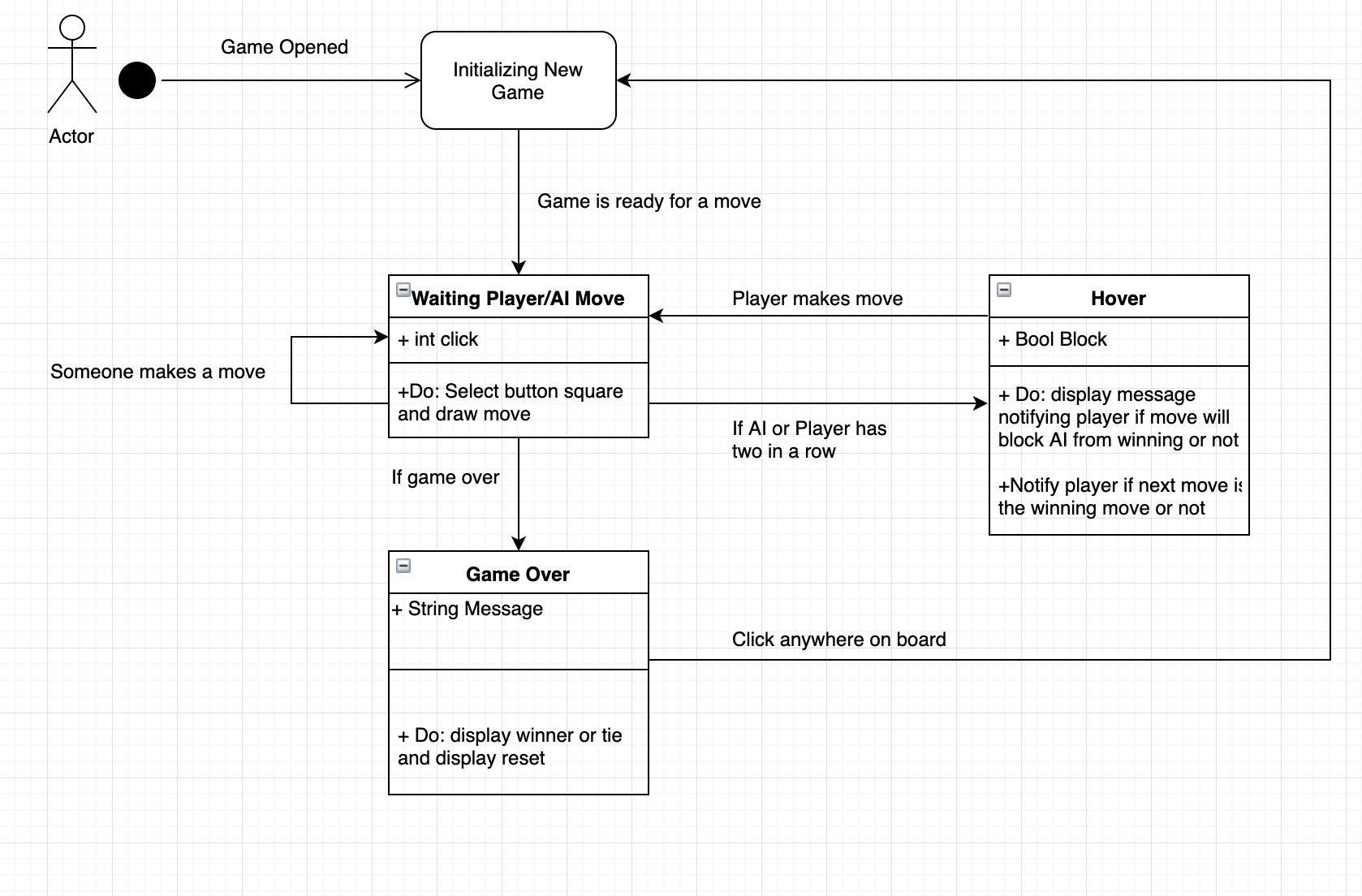
|  |  |
| --- | --- |
| **Class**: Board | |
| **Superclass or Subclass** : | |
| ***Responsibilities*** | ***Collaborators*** |
| *boolean gameover - check to see if the game is done* | *Player* |
| *Button[] allButtons - all of the buttons on the Tic-Tac-Toe board* | *Button* |
| *Player player - holds the state value the player is in* |  |
| *Board() - creates a Tic-Tac-Toe board to play on* |  |
| *resetBoard() - clears board and clears all the buttons on it* |  |
| *initializeButtons() - creates buttons for the Tic-Tac-Toe board* |  |
| *drawLines() - draw lines separating the buttons of the Tic-Tac-Toe board* |  |
| *getUserInput() - see what button the user pressed on the board* |  |
| *aiTurn() - makes a move for the AI on the computer’s turn by using the minimax algorithm* |  |
| *aiTurnMinimax() - goes through the game and keeps track of wins and keeps score of the most efficient way to win the game* |  |
| *findWinForPlayer() - finds the winning move for the player* |  |
| *findOpenSquareForAl() - finds the winning square for the AI* |  |
| *findFork() - Finds a fork for the player to make* |  |
| *blockFork() - if the AI can make a for, tell the player where to go to block it* |  |
| *displayMessage() - display messages to help the user win and advice to block the AI* |  |
| *drawBoard() - draws the Tic-Tac-Toe board* |  |
| *validInput(...) - check to see if the input from the user is valid* |  |
| *returnWinner() - check to see “X” or “O” won* |  |
| *checkGameOver() - Check to see if the game is won by either the player or computer or if it’s a tie* |  |
| *countTurns(...) - count the number of turns that have passed in the game* |  |
| *makeTurn(...) - moderates board turns for the user and the player* |  |
| *getGameover() - returns whether the game is finished or not* |  |
| *getBoardStateString() - returns the buttons and the states for the Tic-Tac-Toe board* |  |
| *getButtonText() - returns the test on each of the buttons* |  |
| *getPlayer() - returns the player* |  |
| *toString() - returns the state of the board as a string* |  |
| *equals(...) - check if two boards are equal* |  |

|  |  |
| --- | --- |
| **Class**: Button | |
| **Superclass or Subclass** : | |
| ***Responsibilities*** | ***Collaborators*** |
| *int xCoordinate - x coordinate of button* | *Board* |
| *int yCoordinate - y coordinate of button* |  |
| *int width - width of button* |  |
| *int height - height of button* |  |
| *String label - name of button* |  |
| *State start - state of button* |  |
| *Button(...) - creates button objects for board* |  |
| *isInside(...) - checks if player cursor is in the button* |  |
| *display() - displays the button on the board* |  |
| *getXCoordinate() - returns x coordinate* |  |
| *getYCoordinate() - returns y coordinate* |  |
| *getWidth() - returns button width* |  |
| *getHeight() - returns button height* |  |
| *getLabel() - return button name* |  |
| *getState() - return button state* |  |
| *setXCoordinate(...) - set x coordinate* |  |
| *setYCoordinate(...) - set y coordinate* |  |
| *setWidth(...) - set button width* |  |
| *setHeight(...) - set button height* |  |
| *setState(...) - set button state* |  |
| *setLabel(...) - set button label* |  |
| *drawShapes() - draws an “X” or an “O” depending on the state of the button on the board* |  |
| *toString() - return a button as a string* |  |
| *equals(...) - check if two buttons are equal* |  |

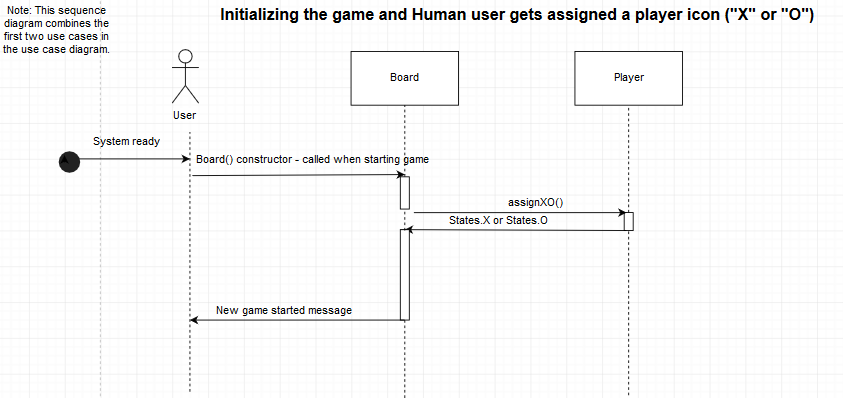
|  |  |
| --- | --- |
| **Class**: Player | |
| **Superclass or Subclass** : | |
| ***Responsibilities*** | ***Collaborators*** |
| *States xo - holds which state the user is in* | *States* |
| *Player() - default player constructor* | *Board* |
| *assignXO() - assigns the player as either “X” or “O”* |  |
| *setXO() - sets the player as “X” or “O” manullely* |  |
| *getXO() - returns what state the player is in* |  |

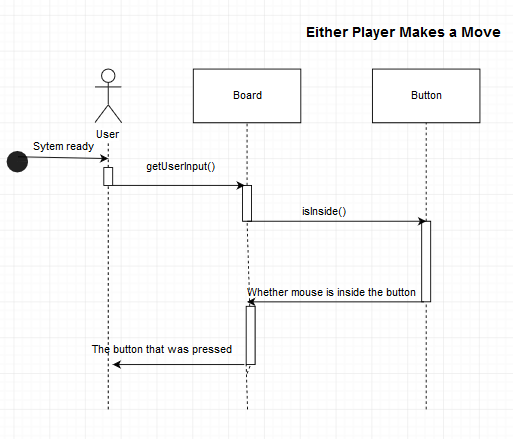
|  |  |
| --- | --- |
| **Class**: States | |
| **Superclass or Subclass** : | |
| ***Responsibilities*** | ***Collaborators*** |
| *Enumeration of States* | *Player* |
|  | *Board* |
|  |  |

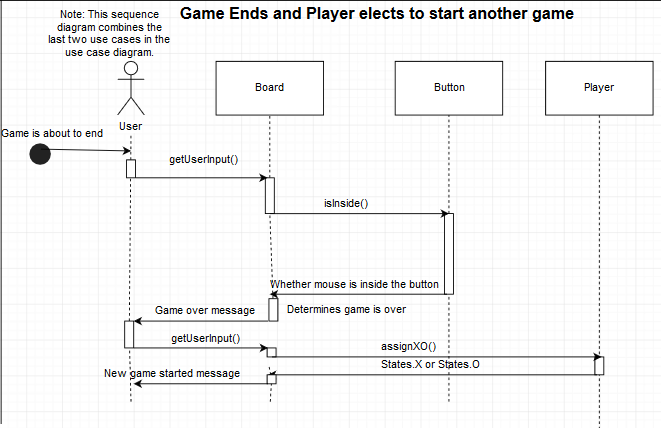
**State Diagram**



**Sequence Diagrams**







**Test Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Name** | **Pre conditions that have to be in place before running this test** | **Input to drive the test** | **Expected outcome** |
| Player goes first at the center. | The player is assigned an X. | Player should make the following moves in this order: center, corner that is not opposite to the corner that the AI made a move, in-between the O squares, empty square next to the center, and the last remaining square | Game ends in a tie. |
| Player goes first at a corner. | The player is assigned an X. | Player should make the following moves in this order: corner, corner that is not opposite to the corner that the player made their first move, next to the two O squares, next to the two O squares | Game ends in a tie. |
| Player goes first at one square next to center. | The player is assigned an X.  The player made a move one square next to center. | Player should make the following moves in this order: one square next to the center, center, next to the center | Game ends in a tie. |
| AI goes first at corner. | The player is assigned an O.  The AI made a move at the top left corner. | Try to create a “fork” scenario and see if the AI can avoid it or if it will try to fork the Player | Game ends in a tie. |
| Check for all empty empty squares that will make player win if they make a move. | The game has started and the player is assigned X or O. | Attempt to play the tic-tac toe game normally. After each new move, check whether there exists an empty space that will allow **the player** to win. Hover over any empty space except that space. A message should be displayed to the player that an empty space exists that allows the player (you) to win. If this message is not displayed, there is a bug. Repeatedly play the game until every possible empty space (3 per row, column, and diagonal) is checked. | A message is shown throughout the game as to which empty spaces will cause the player to win every time a space exists until the game ends in any way. |
| Check for all empty empty squares that will make AI win if they make a move. | The game has started and the player is assigned X or O. | Attempt to play the tic-tac toe game normally. After each new move, check whether there exists an empty space that will allow **the AI** to win. Hover over any empty space except that space. A message should be displayed to the player that an empty space exists that allows the AI to win. If this message is not displayed, there is a bug. Repeatedly play the game until every possible empty space (3 per row, column, and diagonal) is checked. | A message is shown throughout the game when the user hovers a blank square the way as to which empty spaces will cause the player to win every time a space exists until the game ends in any way. |
| Check for all empty empty squares that the player can take to make a fork. | The game has started and the player is assigned X or O. | Attempt to play the tic-tac toe game normally. After each new move, check whether there exists a space that will allow **the player (you)** to make a fork. Hover over any empty space except that space. Repeatedly play the game until every possible fork that the player (you) can make is checked. | A message should be displayed to the player that a space exists that allows the player (you) to make a fork every time one exists throughout the game. |
| Check for all empty empty squares that the AI can take to make a fork. | The game has started and the player is assigned X or O. | Attempt to play the tic-tac toe game normally. After each new move, check whether there exists a space that will allow **the AI** to make a fork. Hover over any empty space except that space. Repeatedly play the game until every possible fork that the player (you) can make is checked. | A message should be displayed to the player that a space exists that allows the AI to make a fork throughout the game. |

**Rationale**

We follow the order by which tasks appeared first, then assign each person with one task at a time. Whoever finishes his task moves on to the next one until all the tasks are finished. When everyone is finished, we will all together review the tasks and improvise.

The difficult part about this assignment is the workload. There are a lot of diagrams that we have to construct, which takes a significant amount of time. Other than that, it was a fun lab!