CONNECT 4

TEAM 5

18TH MARCH 2018

Arushi Gupta, Jay Shah, Swati Sinha, Zhenyu Zhou

Index

1.0. Introduction

1.1 Purpose

The objective is to implement the game Connect 4 while adhering to the Object-Oriented Analysis, Design and Programming principles. It will explain the features of the system and what all this system can do.

1.2 Overview

The product is a software application which is used to play the game Connect 4. It is a two-player game in which the players take turns dropping discs into a seven-column, six-row grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to be the first to form a horizontal, vertical or diagonal line of four discs of the same color.

2.0 UML Diagrams

2.1 Class Diagram

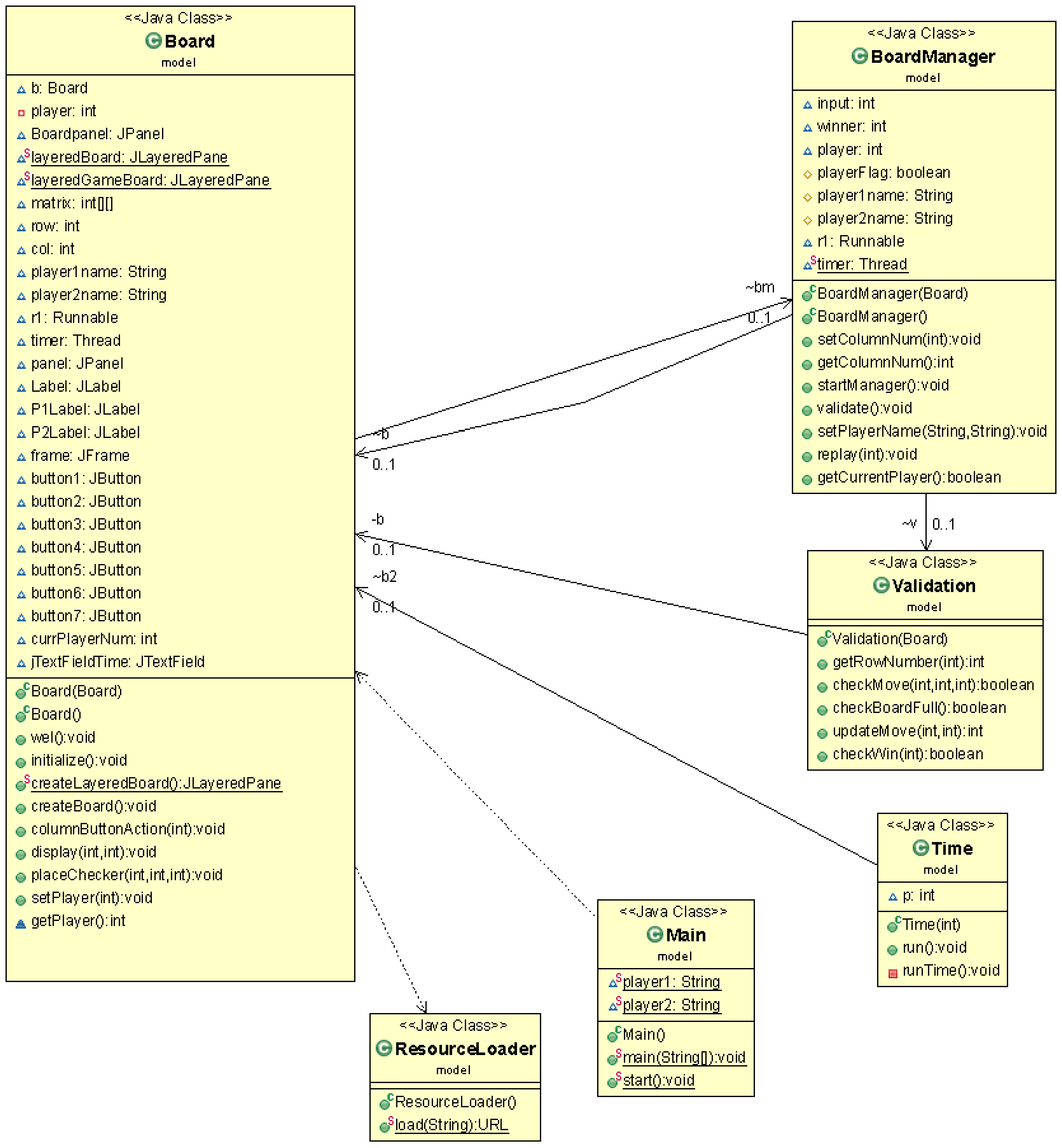


Figure 1: Class Diagram

The various classes that have been used in this game are class Main, Board, BoardManager, Validation, ResourceLoader and Time.

The starting point of the program is class Main which includes public static void main().

Main:

- It consists of main() whose work is to create Board object.

Board:

- It consists of all the GUI stuff such as buttons, panels, labels, layeredpane, frame.
- wel() takes player names as input from the user and stores it.
- initialize() is used to set player names, label, and player turns on the board.
- createLayeredBoard() is used to create a layered board to play discs.
- createBoard() is used to create the buttons board.
- columnButtonAction() is used to start a timer and communicates with the BoardManager that which column is clicked.
- display() displays which player's turn it is on the GUI and prints out time countdown in console.
- placeChecker() places the correct disc on correct position.
- It calls the ResourceLoader class for getting the image of the board and discs.

BoardManager:

- BoardManager manages the class Board and class Validation.
- validate() is used to call Validation class with the current player and the column selected by the player.
- replay() is used to ask the user whether the user wants to replay the game. If user clicks YES, then BoardManager resets the Board, otherwise it quits the game.

Validation:

- getRowNumber() finds the correct row number to place the disc.
- checkMove() is used to check whether the move made by player is within the board.
- checkBoardFull() is used to check whether the Board is full or not.
- updateMove() is used to update the move of the player and inform about it to BoardManager.
- checkWin() is used to check whether there is any winner in the game or not. It checks 4 consecutive discs in a row, horizontally, vertically and diagonally.

ResourceLoader:

- It loads the images of the board and the color which we have used as disc.

Time:

- Threads in Time class is used to get the timer of 5 seconds, the player should move the disc within it, to be in the game, or else the opponent player wins the game.

2.2 Sequence Diagram

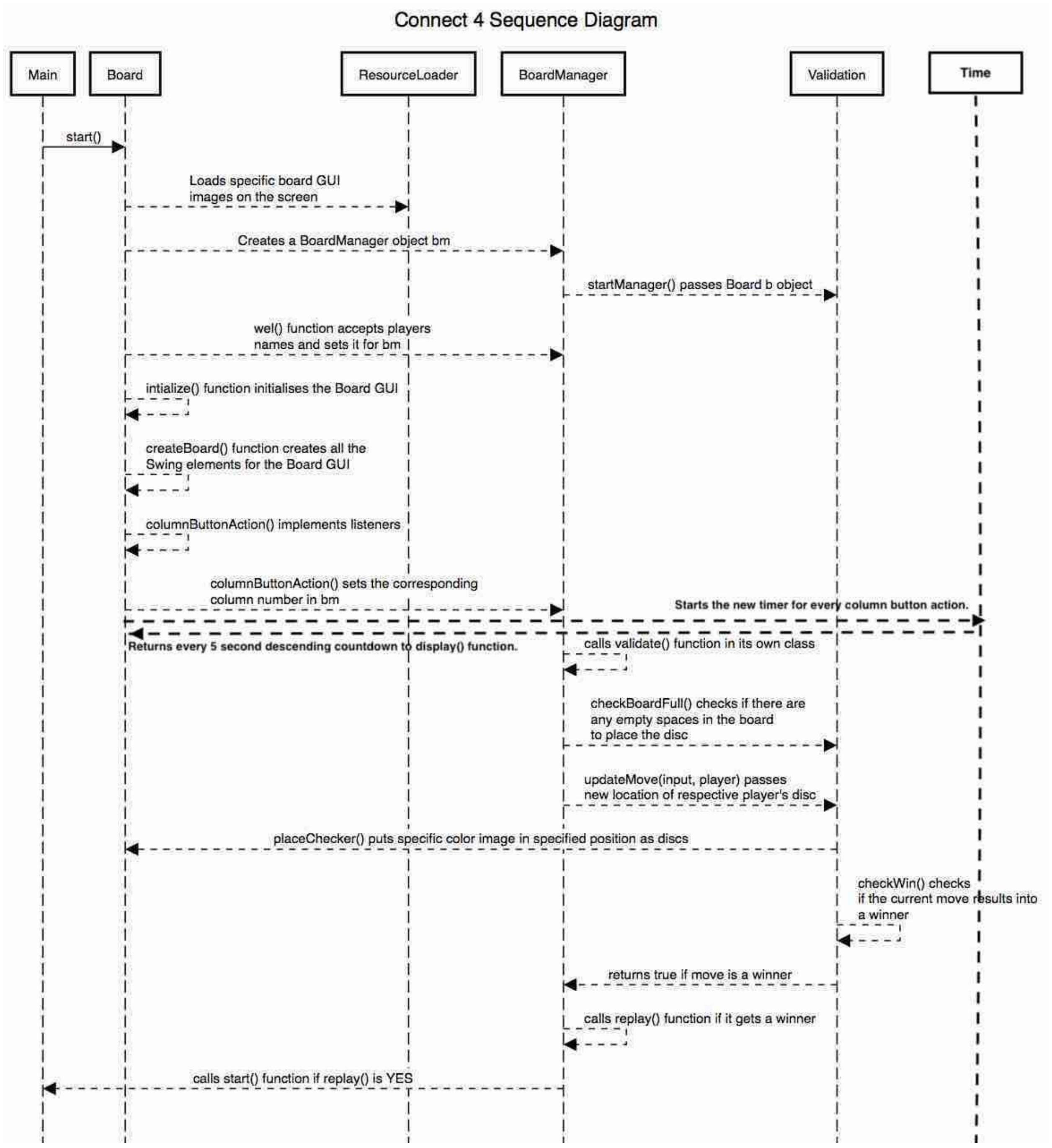

Connect 4 Sequence Diagram
Main
Board
ResourceLoader
BoardManager
Validation
Time
start()
Loads specific board GUI images on the screen
Creates a BoardManager object bm
startManager() passes Board b object
wel() function accepts players names and sets it for bm
intialize() function initialises the Board GUI
createBoard() function creates all the Swing elements for the Board GUI
columnButtonAction() implements listeners
columnButtonAction() sets the corresponding column number in bm
Starts the new timer for every column button action.
Returns every 5 second descending countdown to display() function.
calls validate() function in its own class
checkBoardFull() checks if there are any empty spaces in the board to place the disc
updateMove(input, player) passes new location of respective player's disc
placeChecker() puts specific color image in specified position as discs
checkWin() checks if the current move results into a winner
returns true if move is a winner
calls replay() function if it gets a winner
calls start() function if replay() is YES

Figure 2: Sequence Diagram

Using our class diagrams and sequence diagrams, we started implementing our code for the game Connect 4. We started with creating a Main() class that calls a local start() function which calls the Board class. Now the Board class object initializes the GUI aspect of our game by accepting and displaying the names of the players in our main frame, along with the board image and the column buttons (1-7). ResourceLoader class uploads the images to the Board class every time the Board class calls it. Depending on the specific function call, either the yellow or the red image file is called to display the corresponding player disc on the screen with every valid move.

The wel() function in Board class sets the names of the players and passes it to BoardManager object. When the BoardManager class is called, it initializes a Validation class object.

The initialize() function displays all the GUI elements of the Board class and calls the createBoard() function. We have used LayeredPane to display two different image layer. First image layer is the Board image and the second layer initially is nothing, but once you make a valid move, the corresponding player disc appears on the second layer.

The columnButtonAction() method is the listener for every column button, it calls the BoardManager and then the BoardManager validates the move and checks for overflow and displays accordingly. Simultaneously, with every column button action, the Time class is called to initiate a 5-second timer countdown to give each player fair amount of time to play a move. If any player fails to make a move under 5 seconds, the other player wins automatically. This is done using threads.

Validation class puts a disc on the screen by calling the Board class placeChecker() method. It also provides the result for every move, whether it resulted into a winner. If yes, returns which player won.

BoardManager then calls the replay() method to get the user’s input whether to play again or quit.

2.3 Use Cases

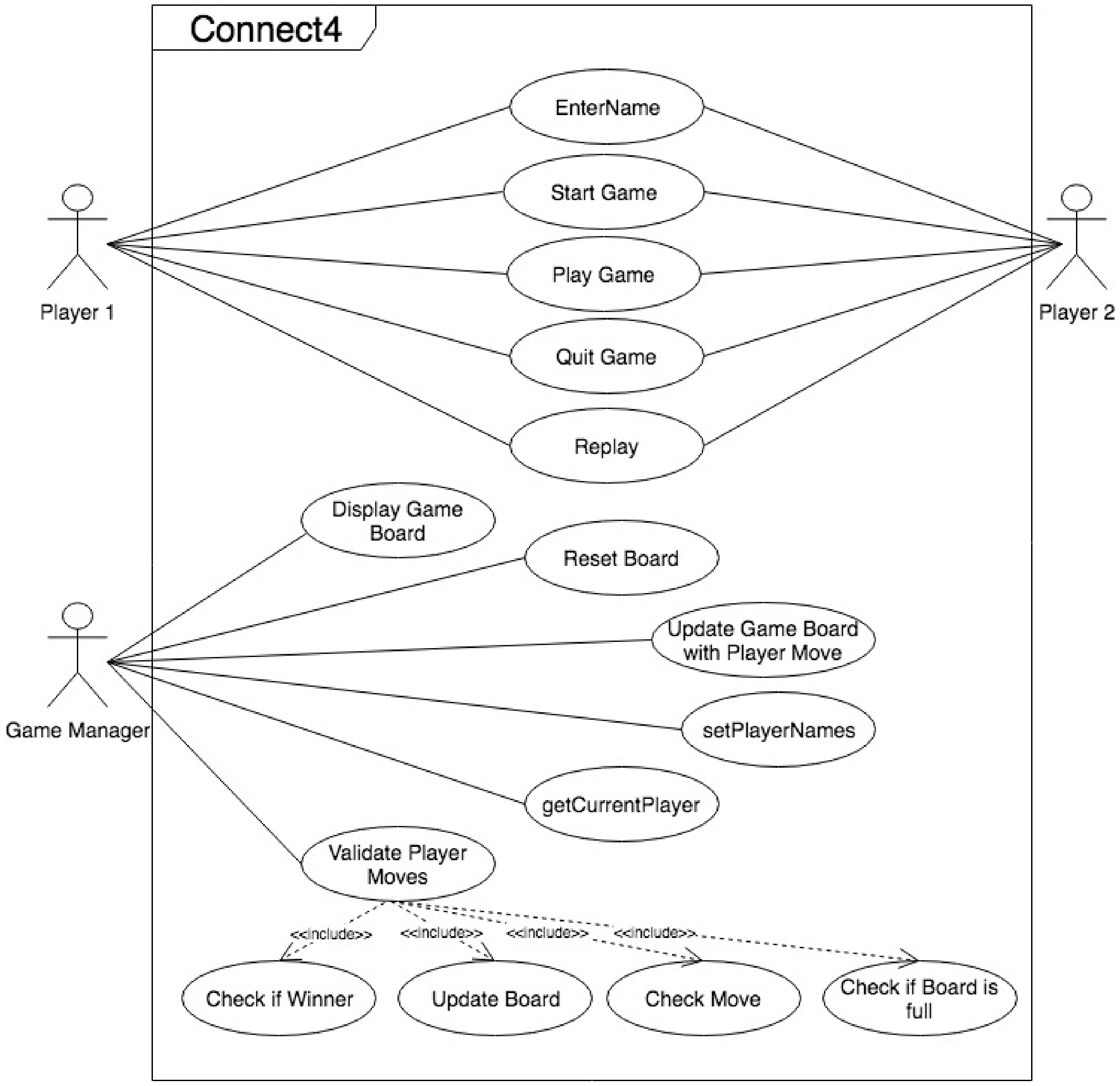


Figure 3: Use Case Diagram

Use case diagram is used for interaction among objects and functions where objects are called actors. We have mapped activities to actors like Player associates with playing, starting, quitting, and replaying the game and the Game Manager manages the game. The Game Manager sets the players' names, updates the game board with the player moves after validating the moves. During validation, it checks if the board is full, if there is a overflow and which player is the winner.

3. Glossary of Terms

Term	Definition
layeredGameBoard	Layered pane that displays the Board image on the screen.
player1name	The Player 1 name that entered in by the user.
player2name	The Player 2 name that entered in by the user.
winner	Player who wins the game by either making four in a row horizontally, vertically, diagonally or if the other player does not play his move in under 5 seconds.
P1Label	Player 1’s name is displayed using this label.
P2Label	Player 2’s name is displayed using this label.
currPlayerNum	Player 1 or 2 that has the current turn.
playerFlag	True or false according to the corresponding player’s turn. True for Player 1’s turn and false for Player 2’s turn
User	Players.

Table 1: Glossary of Terms

4.0. Requirements Specification

4.1 Functional Requirements

- Getting and setting the player names
- Creating the game board
- Determining player's turn
- Validating players' moves
    - Check overflow
    - Check if board is full
    - Check if there is a winner: check horizontally, vertically and diagonally
- Updating the board with the player's moves
- Providing message if there is a winner or the board is full, and providing choices to quit or replay the game

4.2 Non- Functional Requirements

4.2.1 Availability

Availability is the crucial non-functional requirement of the Connect 4 application. Availability can be increased by getting a trusted IDE. The JSwing framework should be displayed every time it is called, and it should not give any error or display something that should not be displayed. This is done using proper exception handlers and using separation of concerns for GUI and data.

4.2.2 Efficiency

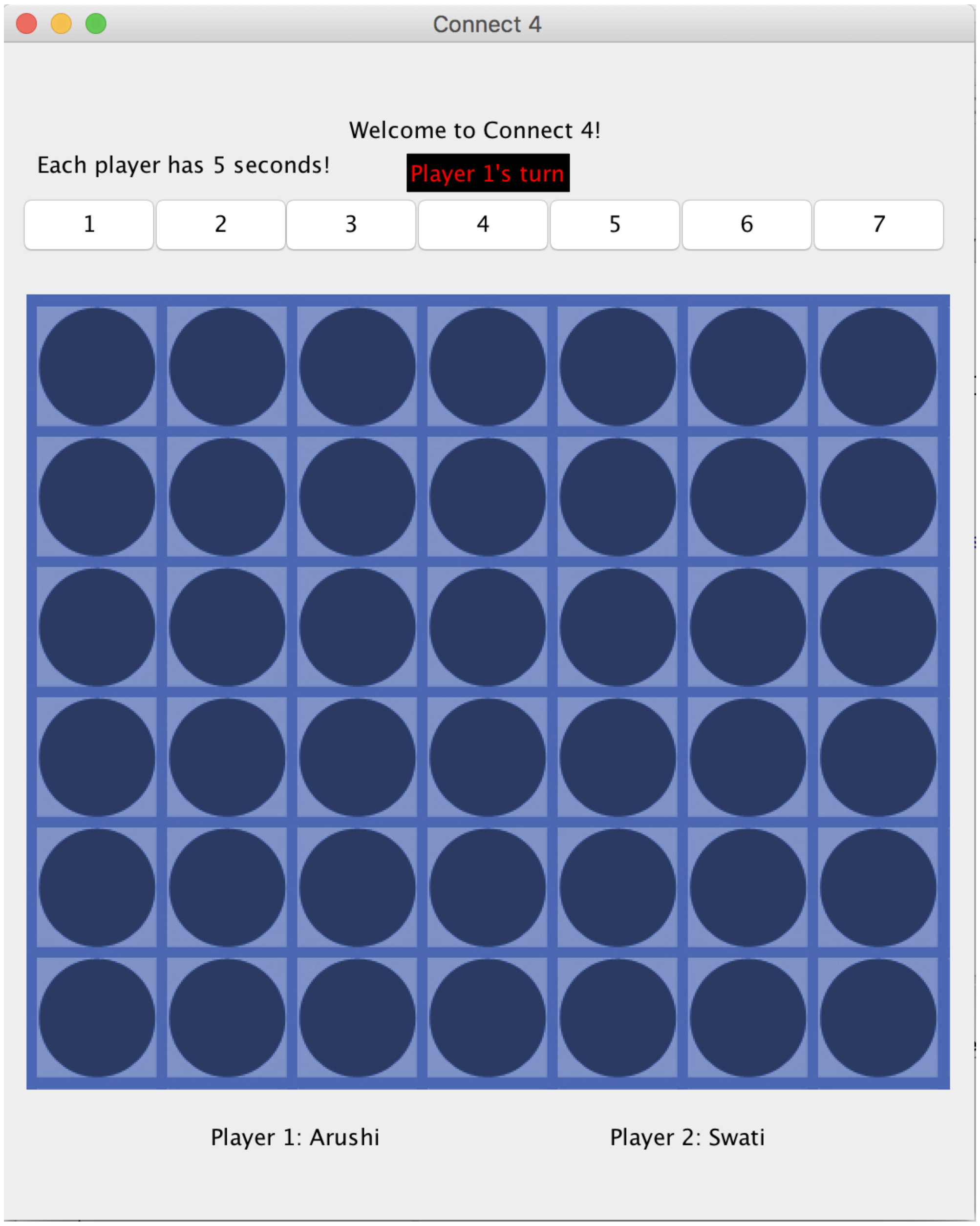
Displaying the appropriate disc (corresponding player disc) is one of the most time consuming process in our application since we need to call the placeChecker() function every time the player clicks the column button(i.e. plays a move). Also, every time a move is played using the column buttons, our code validates every move by checking if there is vacant space left on the board and also checking if that move was a winner move or not. High efficiency can be achieved by choosing most suitable data structure and algorithms to store or write data and calculate the board. Apart from that, errors and wrong result can be avoided in our application by writing efficient code, using proper handlings of exceptions.

4.2.3 Expandability

After the first implementation of the application, we might receive user feedbacks, suggestions, and comments for future improvements. The expandability of the application depends upon factors such as code readability, documentation, QA testing reports which would help in developing any new features in expanding the application. For example, we can make class functions to Undo or Redo a move that the player initially played. Also, we could try adding animation in the movement of the disc when put a disc in a specific column. In this way, the disc appears to fall in that column instead of directly appearing it in the final location.

5.0 Design Prototypes

5.1 Start a new Game



Welcome to Connect 4!

Each player has 5 seconds!

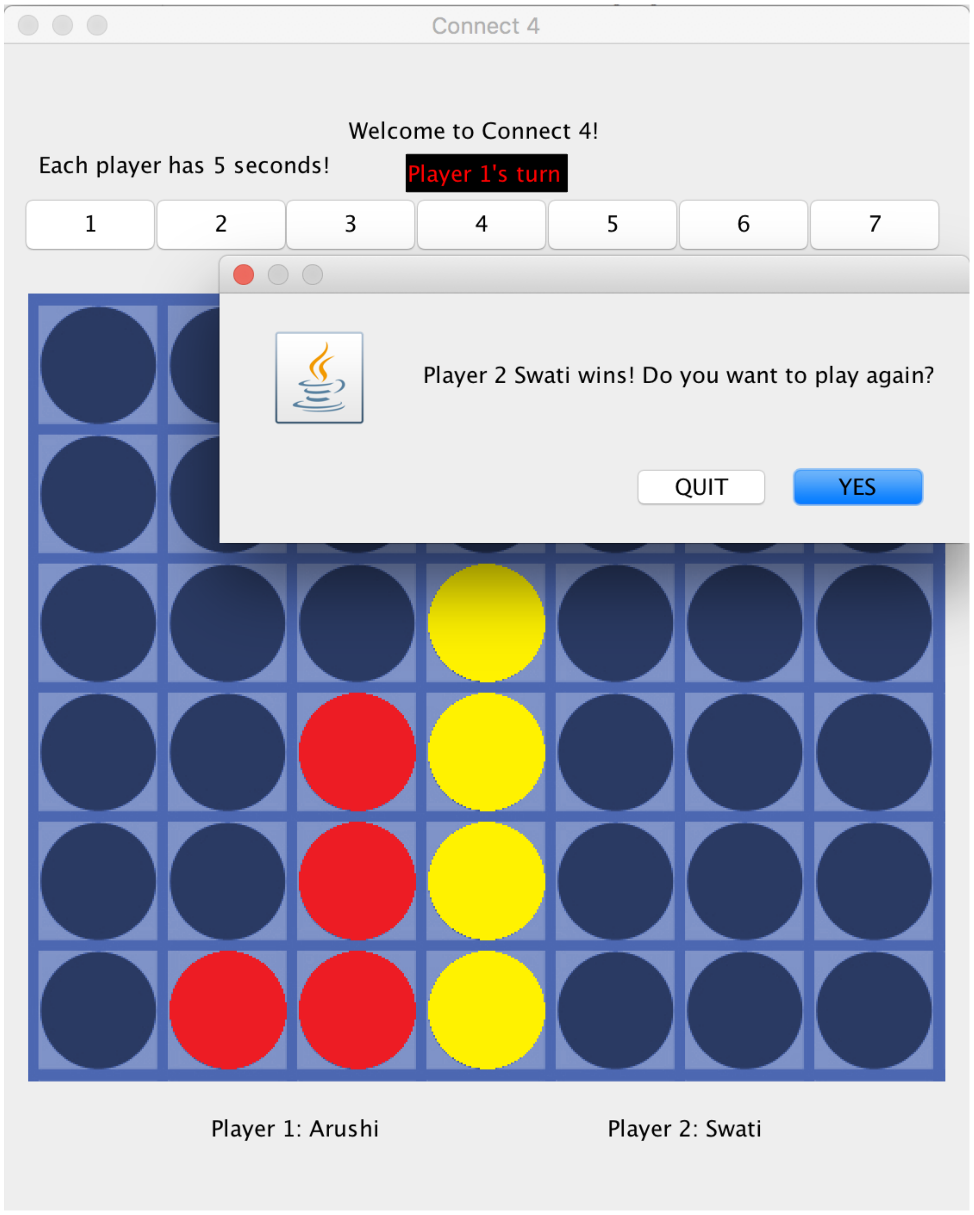
Player 1's turn

Player 1: Arushi

Player 2: Swati

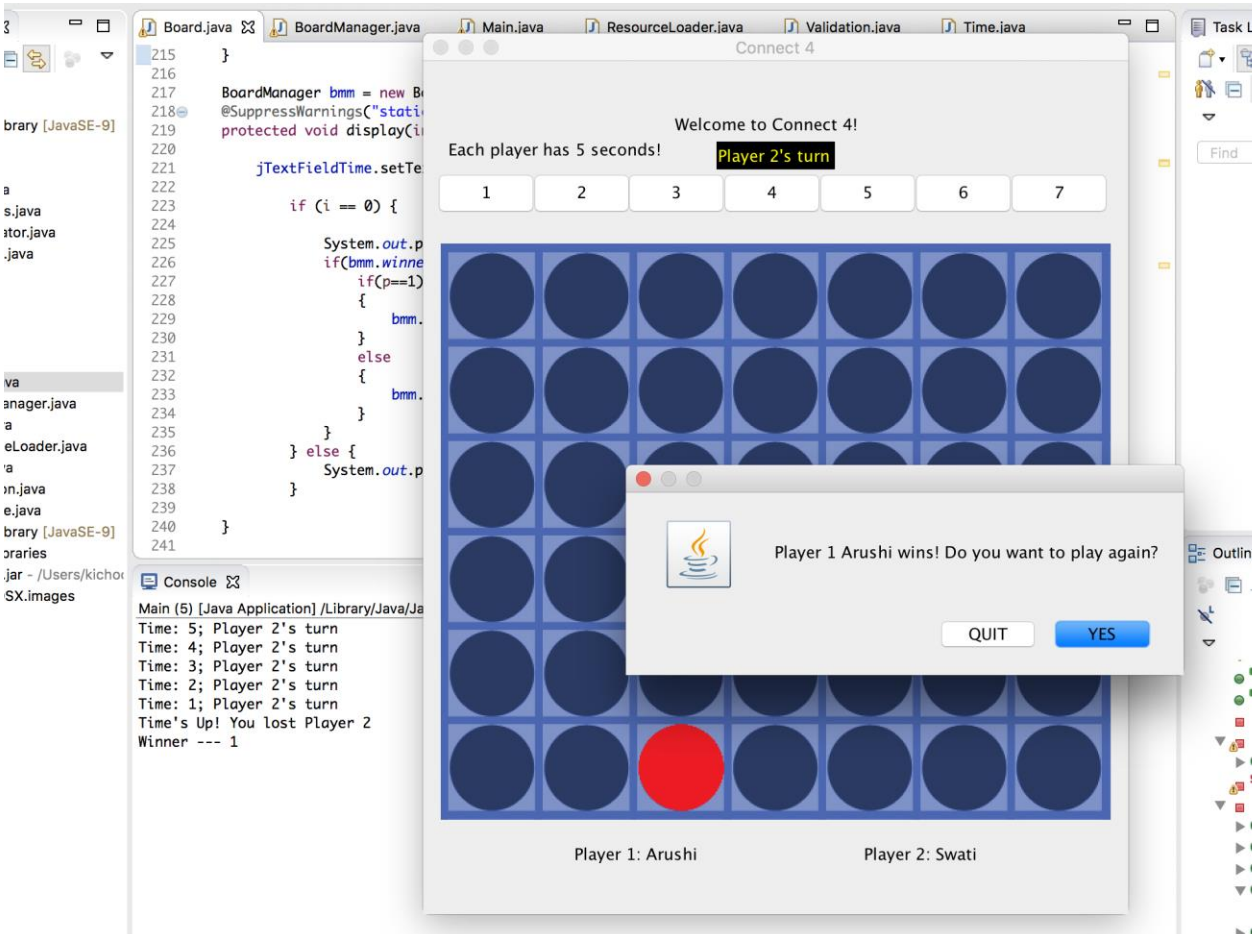
The users enter their names and the names gets stored at the bottom of the screen. The game board has seven options for the user to place discs by clicking on the buttons on the top of the grid. The player whose turn it is gets displayed on the screen.

5.2 Play the Game



A player wins once they get 4 discs of the same color in a row as shown above. Here, Swati won by making four yellow discs in a column. The players can then quit the game or replay.

5.3 Timer



```
    }

    BoardManager bmm = new B
    @SuppressWarnings("stati
    protected void display(i

        jTextFieldTime.setTe

            if (i == 0) {

                System.out.p
                if(bmm.winne
                    if(p==1)
                    {
                        bmm.
                    }
                    else
                    {
                        bmm.
                    }
                }
            } else {
                System.out.p
            }

    }
```

```
Main (5) [Java Application] /Library/Java/Ja
Time: 5; Player 2's turn
Time: 4; Player 2's turn
Time: 3; Player 2's turn
Time: 2; Player 2's turn
Time: 1; Player 2's turn
Time's Up! You lost Player 2
Winner --- 1
```

The timer has been implemented using threads. Each player gets only five seconds to make a move. Once the timer expires, the opponent wins the game. In the given image, Arushi played her move but Swati failed to make a move under 5 seconds which is why Swati lost the game.

6.0 Design Decisions

Analysis phase: It is very important as it creates a rough prototype of our project which at that point does not depend on the technology, platform or syntax.

5 C's: To make it cohesive, complete, consistent, complete and clear.

Law of Demeter: which states that each class or object should have limited knowledge of other classes except the ones that are extended or implemented. The board and validation class have limited knowledge of each other. The board manager is responsible for managing the interaction between these two classes.

Encapsulation: The methods and variables are encapsulated so that the players details are hidden and to separate out the contractual interface between the abstraction and implementation.

Modularity: The program is decomposed into a set of cohesive components and are loosely coupled to minimize dependencies.

Stable intermediate forms: Primitive forms were used to address the problem by understanding the logic and testing it on the console. The simpler problems were solved first. Continuous integration led to more complex systems.

Important design decisions made:

- Simple solutions were used
- Simpler problems were solved first
- Designed throughout the creation of the software
- Had to reshape the problem shape to accommodate new features
- Used an MVC architecture to avoid big ball of mud.

7.0 Test Case Description

For testing, we tested the logic part in the back end and GUI in the front end separately as they have completely different functions. For each of them, we examined every internal structure of program and attempt to test logical case. In other words, we used the unit test to make sure the input causes the output of the program to be identical as the specifications would require. In the end, we integrated all dependent modules together and did the system test to make sure the whole game works properly. We generated several test cases manually.

When we tested our codes, we tested both “good” input, “bad” and malicious mistakes. For example, for the logic part, we checked the boundary conditions and values of different types (i.e. positive, negative) for the game matrix (the 6 * 7 matrix to record the discs locations). In the final product, the user will be able to make invalid move if the column is full but not be able to input negative column number as they are only given valid column number buttons on the screen. We still implemented and test the validation check for both cases to make the code reusable and maintainable. Besides that, we tested the logic based nested statements and case statements. For example, as the board manager class needs to know which player wins the game from validation class, communicates to the board class about it and makes board class place the disc correctly, we tested the nested statements to make sure board class gets the right message and output correctly. In addition, relating to the nested statements, we tested the entering condition and exit values of loop. For GUI, we also made the unit test to make sure everything displays correctly. For example, when the pop-window asks for player 1 and player 2’s names, no matter with or without input, the code runs properly and the pop-up window at the end shows the winner’s name correctly. For the clickable buttons, we also tested the button’s event listener and event handlers separately.

8.0 Summary

In this document, we started off by giving an introduction about the purpose of the product and a general overview. In the second section, we talk about the UML diagrams used. In section three, we gave an overview of the glossary of terms. Section four talks about the functional and nonfunctional requirements. Section five shows the product after it is implemented. Section six talks about the reasons behind the design decisions and finally a test case description for this product is provided in Section seven.