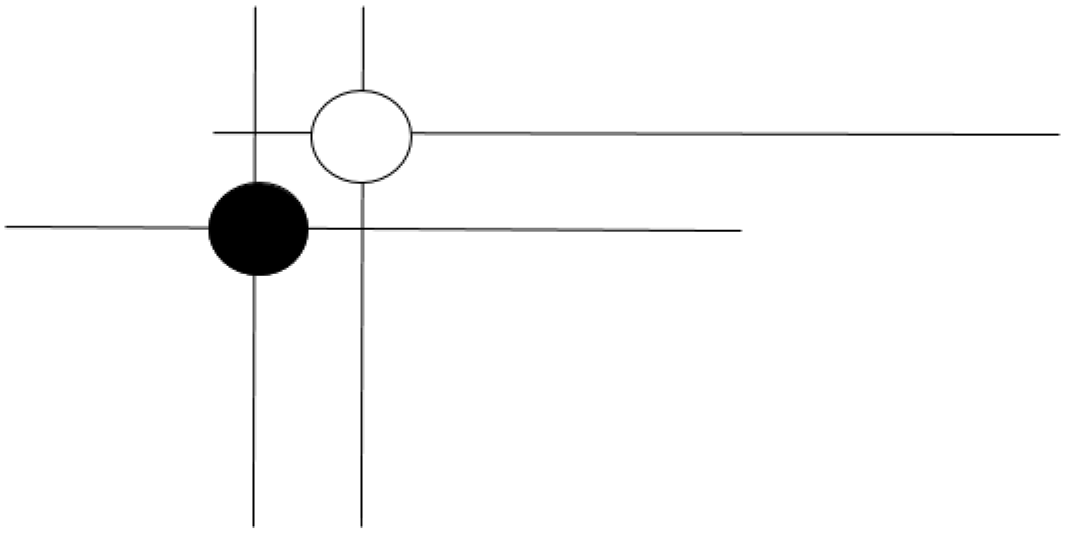


GO
Web Application
Project Overview + Design Process

August 5, 2016
<Version 1.0>

Group Eleven
Zenara Daley | V00820899
Andrew Wiggins | V00817291
Mathew Szymanowski | V00825301
Jinmin Huang | V00787639
Ngoc Thinh Nguyen | V00817304

Table of Contents

1. Introduction

1.1. Purpose

The purpose of milestone 4 is to develop a technical report describing the entire development and implementation process of the Go Web Application. The web application is being developed in partial fulfillment of the requirements of SENG 299 at the University of Victoria.

1.2. Background

There were three preceding milestones that led to the development of the Go Web Application and this report. Milestone 1 was a requirement specification document that described in detail the inputs, behaviors and outputs of the application. Milestone 2 was a software design document which outlined the design of the system and included high level abstractions of the application using UML diagrams. Milestone 3 was a review of a separate groups Milestone 2 document.

1.3. Prerequisites and Structure

This document is intended for individuals directly involved in the development of the Go web application. This includes the software developers and those who will be grading the document and application. Those reading the document should be familiar with JavaScript, CSS, HTML, server-side web development, databases, UML formatting, and the rules of Go. In addition they should familiarize themselves with the “Requirement Specification For Go Web Application” and “Go Web Application Software Design” documents developed by group 11 (milestone’s 1 and 2).

Below is a brief overview of the sections covered.

- : This section describes the purpose, relevant background, and prerequisites for complete understanding of this document.

- : This section offers a summary of the architectural process of the Go project including a high level overview of the system and a review of the original requirements.

- : This describes design patterns used and covers the static, dynamic and allocation views of the Go web application. System, components and modules are modeled and described using UML; also included are descriptions of key algorithms.

- : This describes the timeline of the project, each team member’s roles, problems encountered and solutions.

- : This section covers the review given to our team from milestone three, and how the recommendations changed our design.

- : This sections reflects on the project.

2. Implementation

This section describes the architectural process used in developing and implementing the Go Web Application.

2.1. High Level Overview

The system is a web application that allows the user to play the game Go. The application includes four main user interfaces: start page, login screen, one player game board and two player game board. It is composed of two main components: a client-side application that receives user input, and a server-side application which stores and sends data. The main features of the system currently included in the source code are listed and described as functional requirements below.

2.2. Functional Requirements

This sections lists the functional requirements in a table and indicates whether they were met during implementation; If a requirement was not met, reasoning is included in the comments section. Otherwise, the comment section indicates system elements associated with the requirement. The requirements are listed in order of importance, highest to lowest respectively (see Table 1).

Table 1. Functional Requirement Completion Table.

Requirement	Description	Complete	Comments
Start Page	A user will be prompted to begin a game by choosing either hotseat play or regular play and choose a board size.	✓	The start page and description can be seen in figure 15.
Login Page	In regular play only, this page will prompt a user to login or create a username and password.	✓	The login page and description be seen in figure 16.
Making a Move	A user can click on any open and legal board intersection to place their token. Two options of game play are as follows: - : After move is made, a request will be made to the AI. - : The second user will make a legal move and the AI is not called.	✓	This requirement is addressed in the function "makeMove(x)" as seen in [oneplayer]script.js and [twoplayer]script.js classes (see figure 1). The sequence of events upon making a move can be seen in figures 3 and 4.
External AI	Provided by the instructor. When called, returns a move that is checked for legality then played on the board.	✓	This requirement is addressed in the function "getRandomMove()" in the class AIrequest.js (see figure 2).
Rules	The game will follow the standard rules of Go.	✓	The rules used are linked on the homepage of our application under the tab "rules". The functions

			“check_illegal_move()”, “checkTerr()”, and “capture()” in shared.js abides by these rules (see figure 1).
Scoring	Each player’s score is the value of captured territories minus the number of tokens captured by opponent. A 6.5 point compensation is given to white. Any ambiguities will be handled by the human player.	✓	When tokens are captured, the score is subtracted from the logged score of the opposing colour in the functions “capture(x)” and “removeTokens()”. The “checkTerritory()” function counts the final territories and adds them to each player's score upon the end of a game (see figure 1).
Board Size	Options include 9X9, 13X13, 19X19.	✓	The option to choose a board size is on the start screen UI (see figure 15). The size is set in “setBoard()” in either [oneplayer]script.js or [twoplayer]script.js classes to create the desired board size (see figure 1).
User Accounts	A user's wins will be stored and retrieved by server by providing a screen name.	✓*	On the login page, the user is able to login or create an account, this pushes an object with their username, password and wins to the database using “addAccount()”. This can be retrieved using “getAllAccounts()” on the event of a login (see figure 2). Wins are not updated, see problems section for details.
Undoing a Move	A user can click a button which will revoke the previous move made and allow for a new play.	✓*	This was only implemented in two player mode in the function “undoMove()” in [twoplayer]script.js (see figure 1). See problems section for details about the one player implementation.
Handicaps	Handicaps will provide an advantage for the user by placing four strategic tokens on the board when the game begins.	✓	The option to enable handicaps on the start screen UI (see figure 14). They are enabled in the function “handicap()” in [main]script.js (see figure 1).
Customizable Game Board	The application will allow for the user to change colour themes through a button click during gameplay; a default theme will be given to start.	✓*	The user is given the choice between “Geisha Theme” or “Samurai Theme” on the start page (see figures 17 and 18) rather than during gameplay. No default is given; An alert is given if no theme is chosen. The theme is set in “setBoard()” in either [oneplayer]script.js or [twoplayer]script.js (see figure 1).

2.3. Non Functional Requirements

This sections lists the non-functional requirements in a table and indicates whether they were met during implementation. The requirements of each type are listed in order of importance, highest to lowest respectively (see Table 2).

Table 2. Non-Functional Requirement Completion Table.

Type	Requirement	Description	Complete
Technical	Environment	The source code for the Go application must be written in JavaScript and implemented in a Node JS environment. A Node JS framework may be used.	✓
Technical	Execution	Front end JavaScript code must execute in a web browser. It must not use any frameworks. It may use templating tools or other front-end libraries.	✓
Technical	Databases	A database (if used) must be a MongoDB instance.	✓
Technical	Deployment	The web application must be deployed in a Unix and Windows environment.	✓
Technical	Version Control	Git will be used as a version control system to track and organize code.	✓
Business	Schedule	The application must be completed no later than July 21st.	✓
Quality	Security	Each user account will be password protected.	✓
Quality	Usability	Interface elements (i.e. start page) should be easy to understand.	✓

3. System Architecture

This section dives into the actual architecture of the Go Web Application. Design patterns, views of the system, and key algorithms are described in detail.

3.1. Design Patterns

Both modular and architectural design patterns were used when implementing our Go web application.

2.3.1. Modular

The factory pattern in used in creating our SVG board, grid and tokens. The main script file contains functions that "make" these objects. However, the actual creation logic is contained within a separate SVG factory javascript file.

The decorator pattern is used in the implementation of the go game board. The main javascript file creates a board filled with tokens. The tokens can then be "decorated" with the one player or two player javascript file's functionality the achieve the desired level of customization at runtime. This includes mouseover and onclick functionality which will make the pre existing token objects visible to the client.

The singleton pattern was used to secure a sole connection to the database. One database object is created in our server which is referenced when information is being pushed or retrieved.

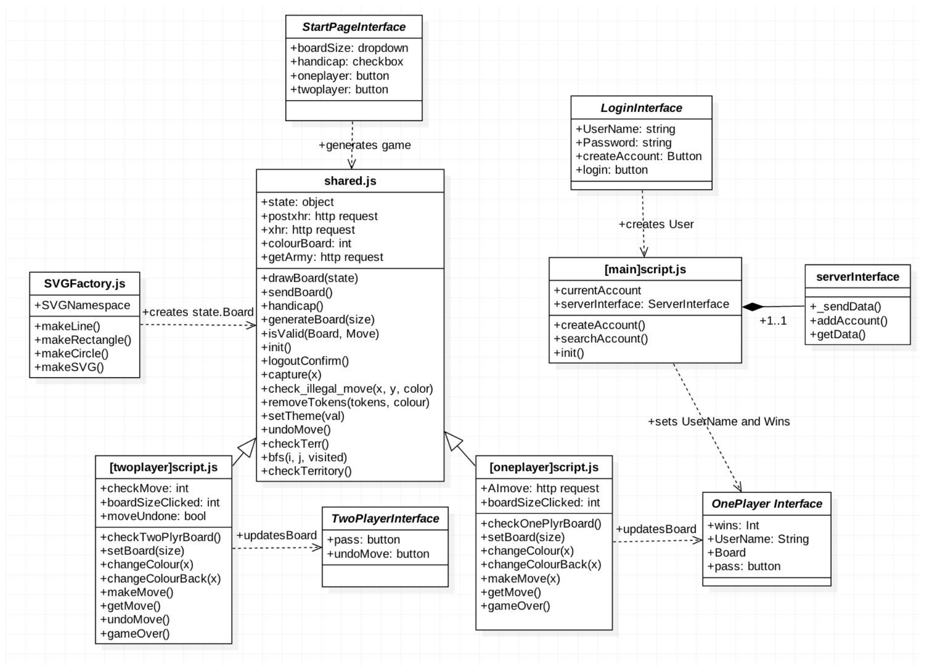
2.3.2. Architectural

The model-view controller pattern was used in designing our Go Web Application. The model in our application encapsulates all of the javaScript files that update and manipulate the application interface. The view includes the four user interfaces of our application: the start page, login page, one player game page and two player game page. The controller is any clickable component of the UI's that manipulate and then update through the model.

3.2. Static Views

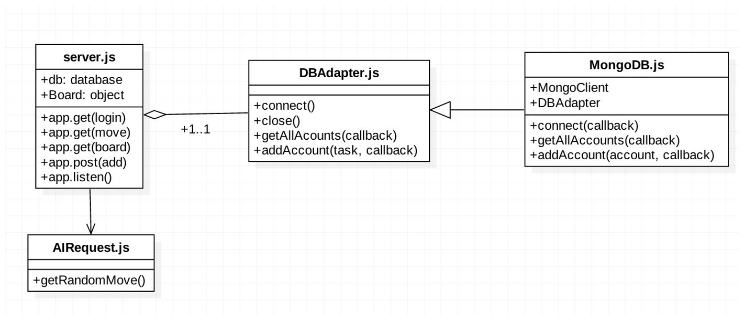
The structure of the front end and back end classes of our Go application has been updated since milestone 2 to meet the requirements of our design (see Figure 1 and Figure 2).

Figure 1. Updated Front End Class Diagram.



The design now has four separate javascript files. One and two player mode scripts hold functionality unique to each game play. The shared script that contains shared functionality of the two, to reduce cloned code. The main script deals with the login UI and sends data such as usernames and wins to the server and database.

Figure 2. Updated Back End Class Diagram.



The server is also now able to make direct requests to the AI. The server has a single connection to the database. The database is an inheritance class of the database adapter.

3.3. Dynamic Views

The structure of the sequence diagrams of our Go application has been updated since milestone 2 to meet the requirements of our design.

Figure 3. Updated One Player Sequence Diagram.

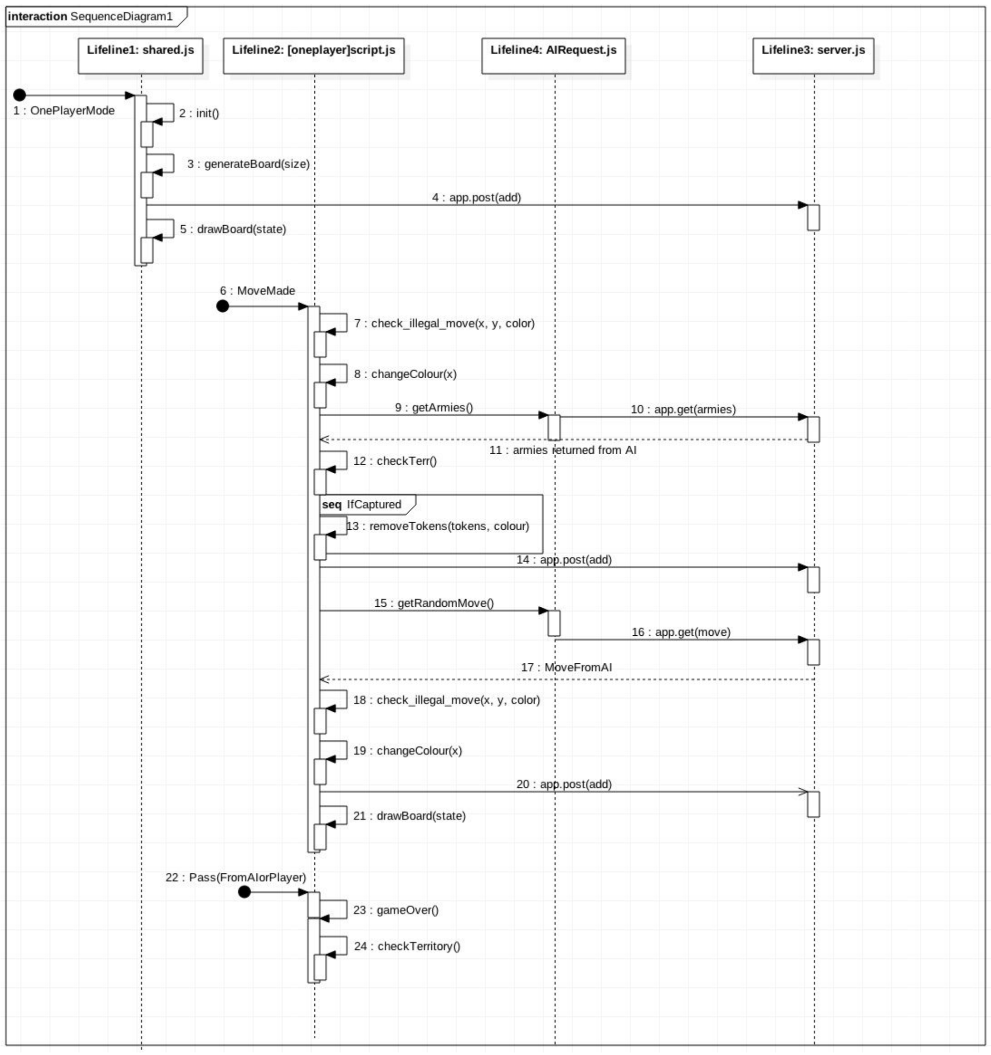
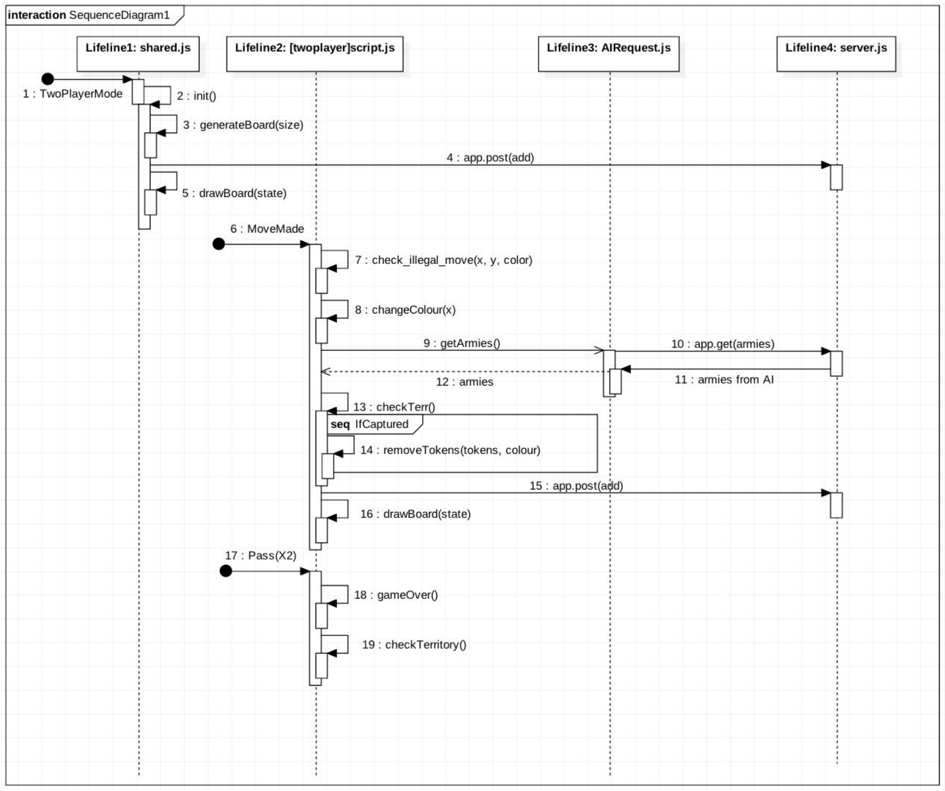


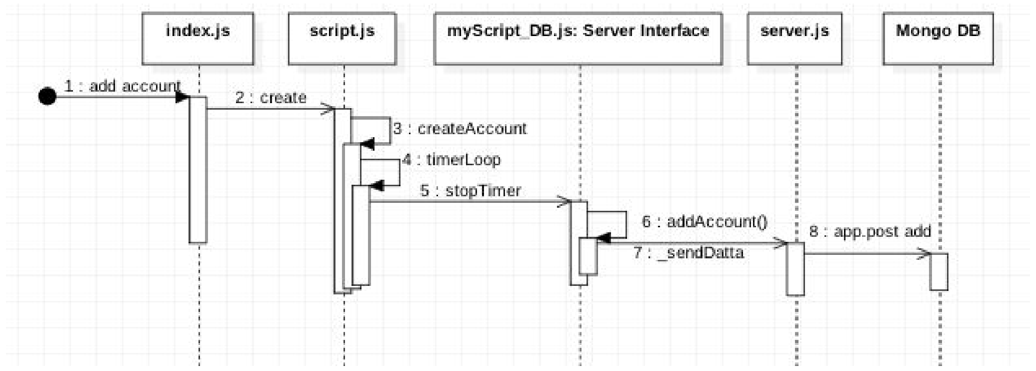
Figure 4. Updated Two Player Sequence Diagram.



In both one player mode and two player mode upon initialization the game board is created, saved in the server and then drawn through SVG. When a move is made, it is checked for legality and updated in the board array. call is then made to the AI's util/armies; upon return, the application checks if any territories have been captured and if so, removes them.

In one player mode a call is then made to the AI's to get a random move. The returned move is checked for legality, added to the board and then finally the board is sent to the server and redrawn. In both modes, if either player makes a pass, the score is calculated and the game ends.

Figure 5. Create Account Sequence Diagram.



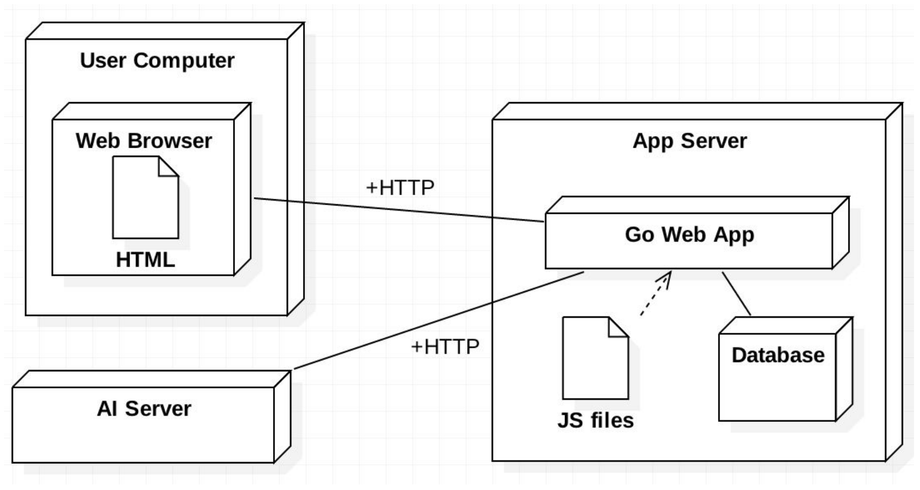
The sequence diagram for creating an account is displayed above. The input of the account details are retrieved from the text field on the HTML page. After the user input is retrieved from clicking the create account button, the create account method pushes the information to the database. The "timerLoop" sets a timeout in case the account creation is unsuccessful.

Logging in uses a similar sequence as above except calls the "searchAccount()" method in [main]script.js (see figure 1). Then a call to get "getAllAccounts()" is made in the server interface.

3.4. Allocation Views

The deployment diagram of our Go application has been updated since milestone 2 to include interaction with the AI (see Figure 6).

Figure 6. Updated Deployment Diagram.



The AI communicates with the App server through HTTP requests, getting moves and information about the location of armies.

3.5. Key Algorithms

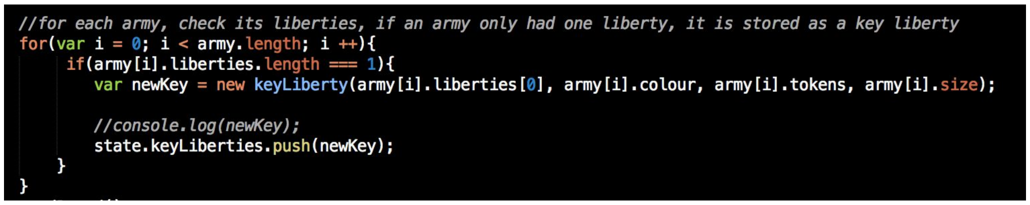
This section describes important functionality within the application's source code.

3.5.1. Removing Captured Territory

: rules, scoring.

In both one player and two player makeMove() functions, a request is made to the AI's 'util/Armies'. Upon a response from the server, all of the armies with only one liberty are stored along with the key liberty and move colour needed for capturing (see figure 7).

Figure 7. Code to store armies with only one liberty.



```
//for each army, check its liberties, if an army only had one liberty, it is stored as a key liberty
for(var i = 0; i < army.length; i ++){
    if(army[i].liberties.length === 1){
        var newKey = new keyLiberty(army[i].liberties[0], army[i].colour, army[i].tokens, army[i].size);

        //console.log(newKey);
        state.keyLiberties.push(newKey);
    }
}
```

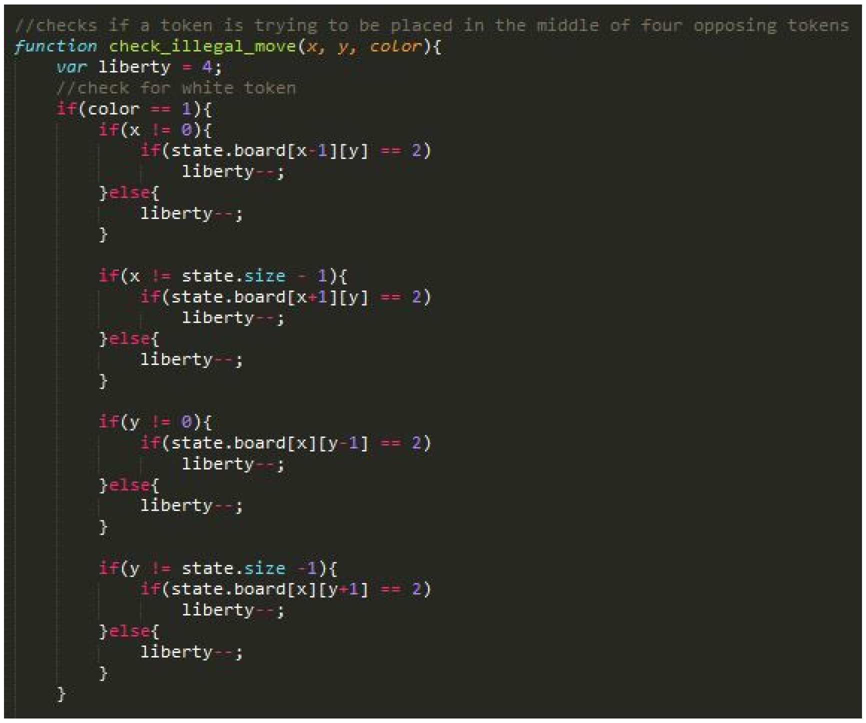
If the key liberty move is made on the next move, then the army is deleted by calling removeTokens() and points are given to the team that captured. Each time a move is made the key liberties are cleared and updated to prevent invalid key liberties from being stored.

3.5.2. Checking For an Illegal Move

: rules.

The code below shows the algorithm for 'check_illegal_move(x, y, color)' (see figure 8).

Figure 8. Code to check illegal move.



```
//checks if a token is trying to be placed in the middle of four opposing tokens
function check_illegal_move(x, y, color){
    var liberty = 4;
    //check for white token
    if(color == 1){
        if(x != 0){
            if(state.board[x-1][y] == 2)
                liberty--;
        }else{
            liberty--;
        }

        if(x != state.size - 1){
            if(state.board[x+1][y] == 2)
                liberty--;
        }else{
            liberty--;
        }

        if(y != 0){
            if(state.board[x][y-1] == 2)
                liberty--;
        }else{
            liberty--;
        }

        if(y != state.size -1){
            if(state.board[x][y+1] == 2)
                liberty--;
        }else{
            liberty--;
        }
    }
```

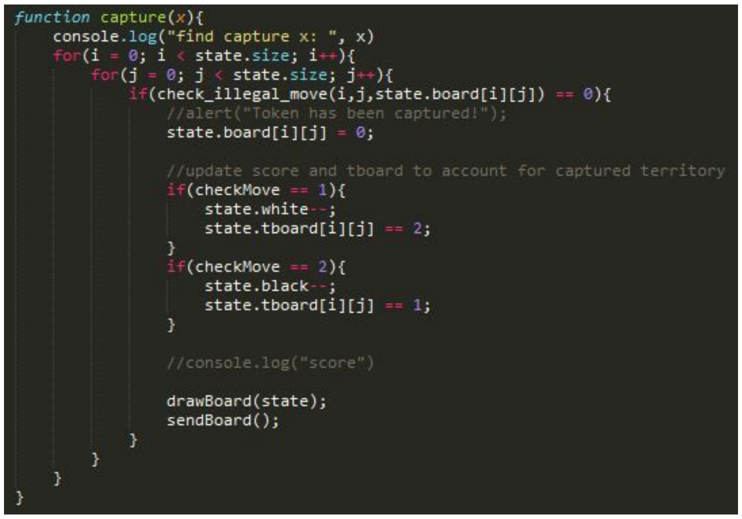
If a token is placed, the four adjacent edges are checked. If all surrounding edges contain tokens of the opposing team, the move is illegal.

3.5.3. Capturing a Token

: rules, scoring.

The code below shows the algorithm for 'capture(x)' (see figure 9).

Figure 9. Code for capturing a token.



```
function capture(x){
    console.log("find capture x: ", x)
    for(i = 0; i < state.size; i++){
        for(j = 0; j < state.size; j++){
            if(check_illegal_move(i,j,state.board[i][j]) == 0){
                //alert("Token has been captured!");
                state.board[i][j] = 0;

                //update score and tboard to account for captured territory
                if(checkMove == 1){
                    state.white--;
                    state.tboard[i][j] == 2;
                }
                if(checkMove == 2){
                    state.black--;
                    state.tboard[i][j] == 1;
                }

                //console.log("score")

                drawBoard(state);
                sendBoard();
            }
        }
    }
}
```

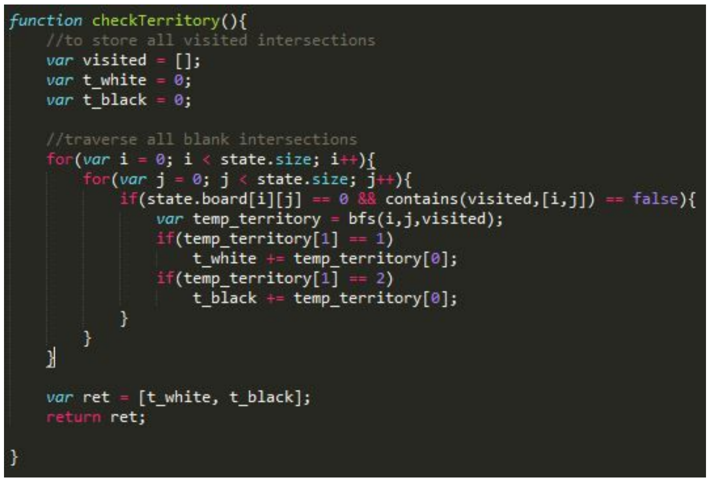
When a token is placed, the capture function will call check_illegal_move to determine if it's adjacent edges are surrounded. If so, the token is removed from the board and points are awarded to the capturer; otherwise it is drawn onto the board.

3.5.4. Checking Territory

: scoring.

At the end of the game the the territory occupied is calculated and then added to the players' scores through 'checkTerritory()' (see figure 10).

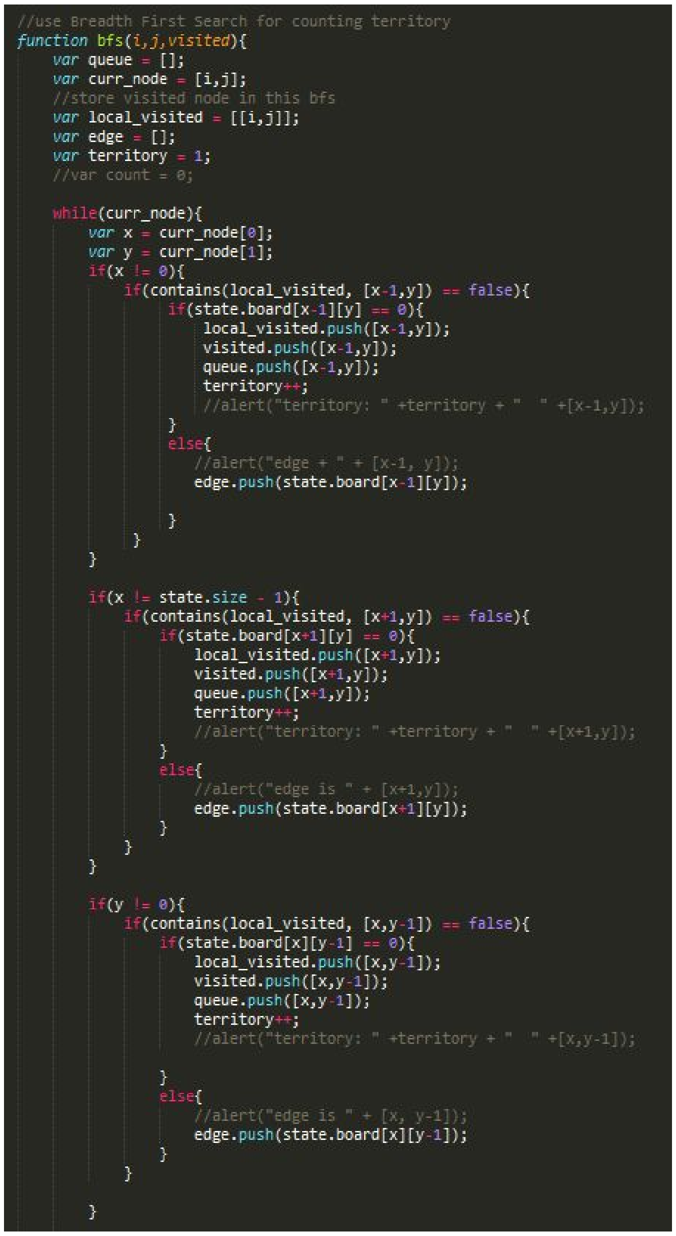
Figure 10. Code for checkTerritory().



```
function checkTerritory(){
    //to store all visited intersections
    var visited = [];
    var t_white = 0;
    var t_black = 0;

    //traverse all blank intersections
    for(var i = 0; i < state.size; i++){
        for(var j = 0; j < state.size; j++){
            if(state.board[i][j] == 0 && contains(visited,[i,j]) == false){
                var temp_territory = bfs(i,j,visited);
                if(temp_territory[1] == 1)
                    t_white += temp_territory[0];
                if(temp_territory[1] == 2)
                    t_black += temp_territory[0];
            }
        }
    }

    var ret = [t_white, t_black];
    return ret;

}
```

First, the function will traverse all intersections on the board. If the intersection is blank and has never been visited, we do breadth first search for this intersection.

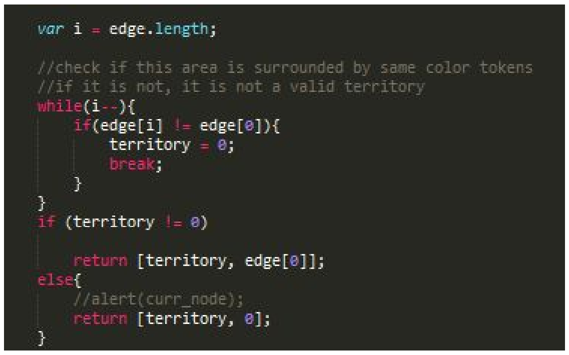
Figure 11. Partial code of breadth first search (main algorithm for bfs).



```
//use Breadth First Search for counting territory
function bfs(i,j,visited){
    var queue = [];
    var curr_node = [i,j];
    //store visited node in this bfs
    var local_visited = [[i,j]];
    var edge = [];
    var territory = 1;
    //var count = 0;

    while(curr_node){
        var x = curr_node[0];
        var y = curr_node[1];
        if(x != 0){
            if(contains(local_visited, [x-1,y]) == false){
                 if(state.board[x-1][y] == 0){
                     local_visited.push([x-1,y]);
                     visited.push([x-1,y]);
                     queue.push([x-1,y]);
                     territory++;
                     //alert("territory: " +territory + "  " +[x-1,y]);
                 }
                 else{
                    //alert("edge + " + [x-1, y]);
                    edge.push(state.board[x-1][y]);

                 }
            }
        }

        if(x != state.size - 1){
            if(contains(local_visited, [x+1,y]) == false){
                if(state.board[x+1][y] == 0){
                    local_visited.push([x+1,y]);
                    visited.push([x+1,y]);
                    queue.push([x+1,y]);
                    territory++;
                    //alert("territory: " +territory + "  " +[x+1,y]);
                }
                else{
                    //alert("edge is " + [x+1,y]);
                    edge.push(state.board[x+1][y]);
                }
            }
        }

        if(y != 0){
            if(contains(local_visited, [x,y-1]) == false){
                if(state.board[x][y-1] == 0){
                    local_visited.push([x,y-1]);
                    visited.push([x,y-1]);
                    queue.push([x,y-1]);
                    territory++;
                    //alert("territory: " +territory + "  " +[x,y-1]);

                }
                else{
                    //alert("edge is " + [x, y-1]);
                    edge.push(state.board[x][y-1]);
                }
            }

        }
```

In bfs(), whenever an end is met, the token is pushed into an array. After the search, the function checks whether all of the stored values are the same. If so, this territory is returned with the corresponding colour; otherwise the territory is reset to zero.

Figure 12. Partial code of breadth first search (return territory and color).


```
var i = edge.length;

//check if this area is surrounded by same color tokens
//if it is not, it is not a valid territory
while(i--){
    if(edge[i] != edge[0]){
        territory = 0;
        break;
    }
}
if (territory != 0)

    return [territory, edge[0]];
else{
    //alert(curr_node);
    return [territory, 0];
}
```

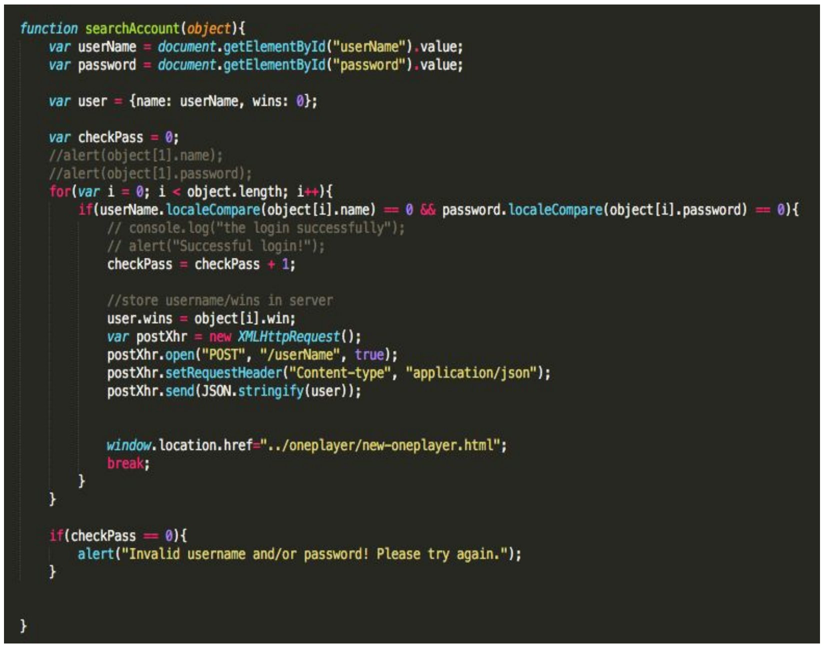
Finally, the checkTerritory() function will take the return value of bfs function. These values are then added to the scores of each team.

3.5.5. Search Accounts

: login page, user accounts.

The code below showcases how a username is retrieved for logging in after it has been stored in the database (see figure 13).

Figure 13. Code for searchAccount().

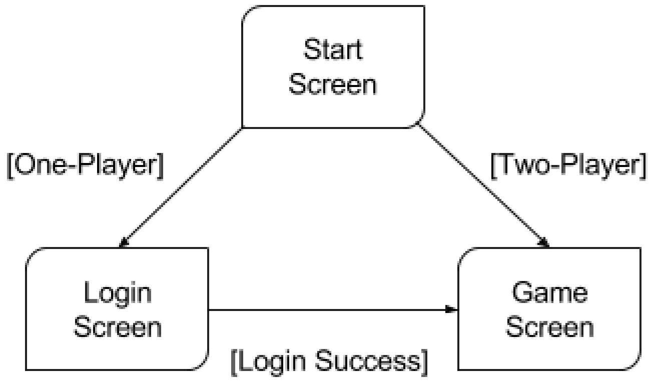

```
function searchAccount(object){
    var userName = document.getElementById("userName").value;
    var password = document.getElementById("password").value;

    var user = {name: userName, wins: 0};

    var checkPass = 0;
    //alert(object[1].name);
    //alert(object[1].password);
    for(var i = 0; i < object.length; i++){
        if(userName.localeCompare(object[i].name) == 0 && password.localeCompare(object[i].password) == 0){
            // console.log("the login successfully");
            // alert("Successful login!");
            checkPass = checkPass + 1;

            //store username/wins in server
            user.wins = object[i].win;
            var postXhr = new XMLHttpRequest();
            postXhr.open("POST", "/userName", true);
            postXhr.setRequestHeader("Content-type", "application/json");
            postXhr.send(JSON.stringify(user));


            window.location.href="../oneplayer/new-oneplayer.html";
            break;
        }
    }

    if(checkPass == 0){
        alert("Invalid username and/or password! Please try again.");
    }


}
```

Upon clicking the login button, a request to retrieve all usernames and passwords is sent to the database through the server. The username and password in the input boxes are then compared with the entries in the database. If there is a match then the login is successful, otherwise the user is prompted to try again.

4. User Interfaces

This section highlights and describes each user interface. Bootstrap framework was used for front end development to help with the overall design quality. The state diagram for the navigation through the user interfaces can be seen below (figure 14).

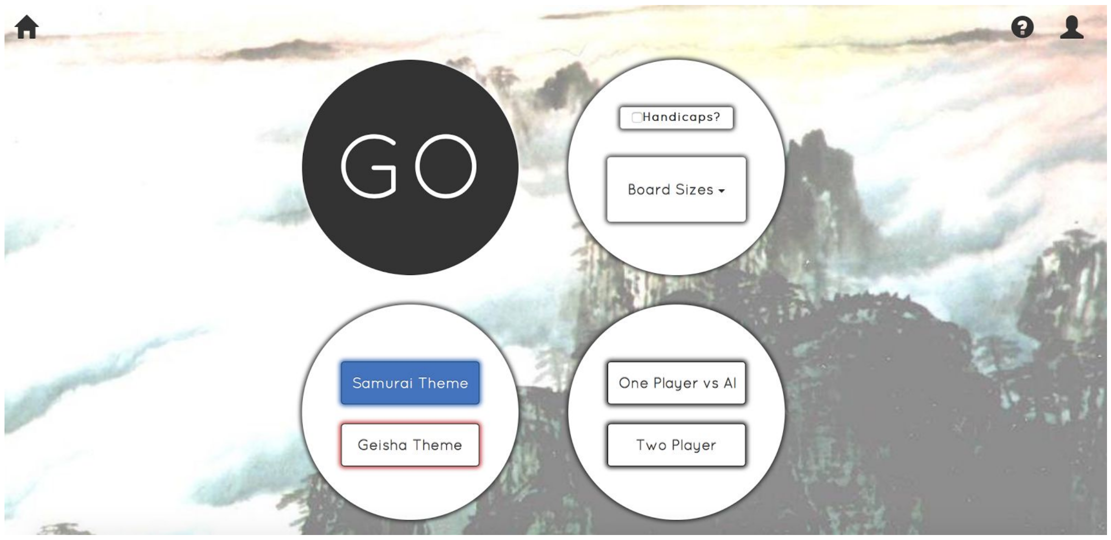
Figure 14. State Diagram for UI Navigation.



4.1. Start Screen

Upon entering the app, the user is prompted to choose a board size, enable or disable handicaps, choose a theme, and choose the gameplay type (see figure 15).

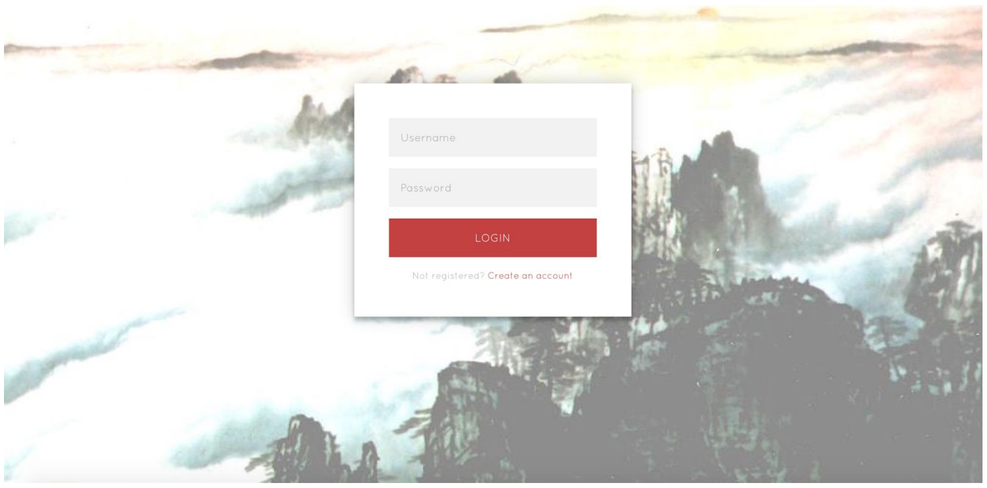
Figure 15. Start Screen User Interface.



4.2. Login Screen

If the user is playing on one player mode, they are directed to a login screen (see figure 16).

Figure 16. Login Screen User Interface.

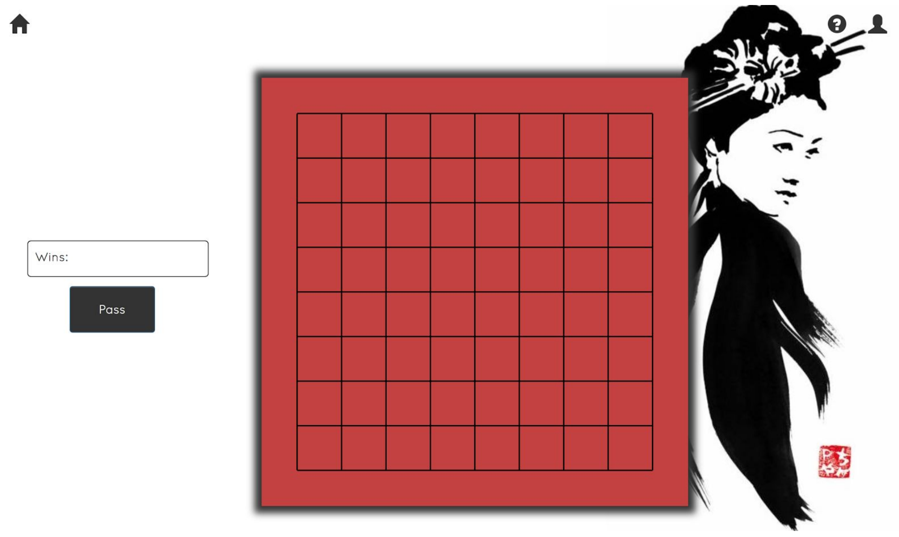


Here the user is able to create an account that is stored in the database or login with a pre existing username and password.

4.3. Game Screen

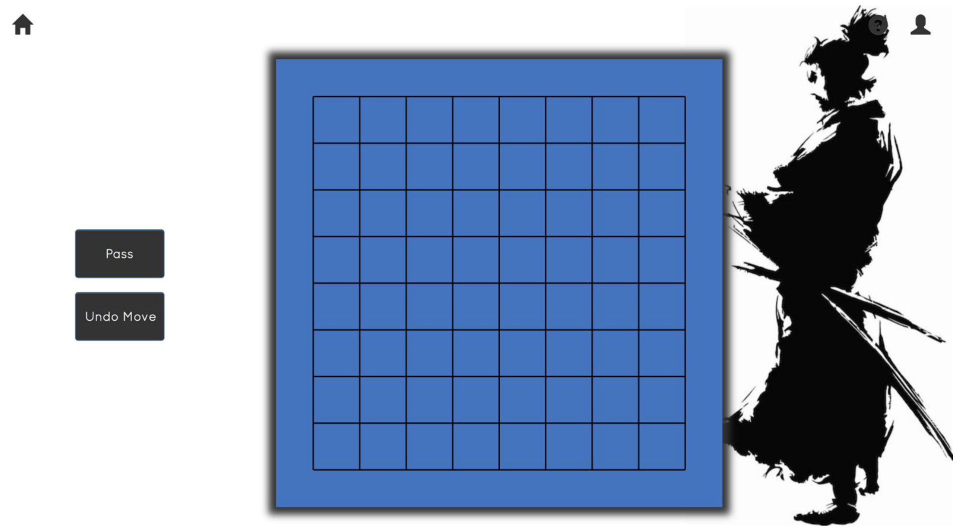
There is a single player game screen and a two player game screen and two themes for each (see figures 17 and 18).

Figure 17. One Player Mode User Interface (Geisha Theme).



In one player mode the user's wins are displayed to the left of the game board, with the pass button located underneath.

Figure 18. Two Player Mode User Interface (Samurai Theme).



In two player mode, there is a pass button and undo move button to the left of the game board.

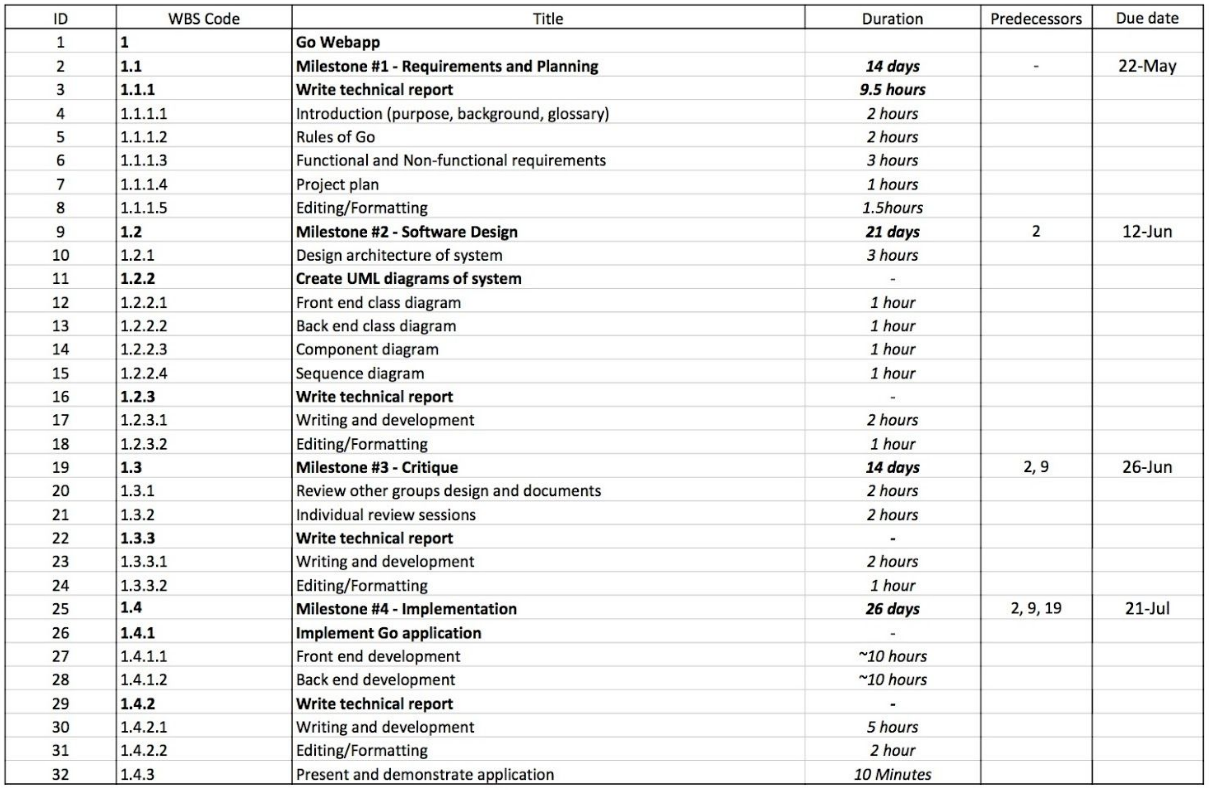
5. Design Process

This section describes the development stages of the project and each member's roles. In addition it covers problems encountered during the project and how they were resolved.

5.1. Project Timeline

The timeline for the Go Web App is shown in table 3.

Table 3. Timeline for Go Web Application Project



ID	WBS Code	Title	Duration	Predecessors	Due date
1	**1**	**Go Webapp**			
2	**1.1**	**Milestone #1 - Requirements and Planning**	***14 days***	-	22-May
3	**1.1.1**	**Write technical report**	***9.5 hours***		
4	1.1.1.1	Introduction (purpose, background, glossary)	*2 hours*		
5	1.1.1.2	Rules of Go	*2 hours*		
6	1.1.1.3	Functional and Non-functional requirements	*3 hours*		
7	1.1.1.4	Project plan	*1 hours*		
8	1.1.1.5	Editing/Formatting	*1.5hours*		
9	**1.2**	**Milestone #2 - Software Design**	***21 days***	2	12-Jun
10	1.2.1	Design architecture of system	*3 hours*		
11	**1.2.2**	**Create UML diagrams of system**	-		
12	1.2.2.1	Front end class diagram	*1 hour*		
13	1.2.2.2	Back end class diagram	*1 hour*		
14	1.2.2.3	Component diagram	*1 hour*		
15	1.2.2.4	Sequence diagram	*1 hour*		
16	**1.2.3**	**Write technical report**	-		
17	1.2.3.1	Writing and development	*2 hours*		
18	1.2.3.2	Editing/Formatting	*1 hour*		
19	**1.3**	**Milestone #3 - Critique**	***14 days***	2, 9	26-Jun
20	1.3.1	Review other groups design and documents	*2 hours*		
21	1.3.2	Individual review sessions	*2 hours*		
22	**1.3.3**	**Write technical report**	**-**		
23	1.3.3.1	Writing and development	*2 hours*		
24	1.3.3.2	Editing/Formatting	*1 hour*		
25	**1.4**	**Milestone #4 - Implementation**	***26 days***	2, 9, 19	21-Jul
26	**1.4.1**	**Implement Go application**	-		
27	1.4.1.1	Front end development	*~10 hours*		
28	1.4.1.2	Back end development	*~10 hours*		
29	**1.4.2**	**Write technical report**	**-**		
30	1.4.2.1	Writing and development	*5 hours*		
31	1.4.2.2	Editing/Formatting	*2 hour*		
32	1.4.3	Present and demonstrate application	*10 Minutes*		

The project was broken down into 4 main milestones, each of which had a specific due date. During milestone 1, we developed our functional and nonfunctional requirements that can be seen in tables 1 and 2. In milestone two, we designed the system using UML, design patterns, and high level abstractions of the system. During milestone three our design was critiqued and recommendations were made that we implemented in our final design (see section 6.1 and 6.2). All milestones were successfully completed and turned in on time.

5.2. Group Member Roles

Each team member took on a specific role and was appointed specific tasks during each milestone (see table 4).

Table 4. Group Members and Their Appointed Roles/Tasks for Each Milestone.

Name	M1 Roles/Tasks	M2 Roles/Tasks	M3 Roles/Tasks	M4 Roles/Tasks
Zenara Daley	Project manager, documentation review	Project manager, documentation review, checkMove() algorithm logic and design	Moderator/scribe, Scheduled meetings, lead review discussions; logged review.	Project Manager, external AI implementation, undo move, removing captured territory.
Andrew Wiggins	Developer and designer	UI development and design. undoMove() implementation.	Scribe/Reviewer, Logged review; checked milestone for defects.	Main UI design and implementation; Bootstrap framework integration; debugging.
Mathew Szymanowski	Developer and designer	Responsible for server development, and implementing the undoMove() method	Scribe/Reviewer, Logged review; checked milestone for defects.	Game board and pass button implementation, assisted with debugging and various methods.
Jinmin Huang	Developer and designer	Front end functions design and implementation.	Scribe/Author, Illuminated unclear ideas; checked milestone for defects.	Login page, illegal move, capture, game over, and check territory functions.
Ngoc Thinh Nguyen	Data analyst, implementation web application	Database development (back end), scoreCount(),	Scribe/Author, Illuminated unclear ideas; checked milestone for defects.	Login page, user accounts implementation.

Roles were determined based on interest and areas of expertise. Each team member contributed an even amount of time and effort to the web application.

5.3. Previous Design

Our previous design outlined in milestone 2 had to be altered slightly to meet the needs of this project (see milestone 2 for original design specification). There were two major changes made: breaking down the main front end class diagram into separate files to reduce coupling between between one and two player game mode and removing the use of the flyweight pattern since it added unnecessary overhead. All of the requirements outlined in milestone two were not altered.

5.4. Problems

This section highlights problems that we encountered during implementation and discusses how we resolved or plan to resolve them.

5.4.1. Availability

One core problem with our project is that it relies heavily on Simon's Go AI for both one-player mode and two-player 'hot seat' mode. Since both modes utilize functionality from the hosted AI, if it were to go down for any reason, our web application would be entirely unplayable. If we were to re-implement the application, we would use the bfs algorithm to not only count territory but also to remove captured territory. An algorithm based remove territory function would make the application available during two player mode even when the AI is down.

5.4.2. Scalability

Our current server and database communication process does not allow for more than one player to be active at a time. In other words, the program cannot write and update more than one game board at any given time. To fix this, we would store the entire game state in the database rather than the server which would allow for multiple games to be played at once.

5.4.3. Updating Wins

One of the requirement features that was originally listed was that a user's win statistics would be stored in the database and listed next to the board during the one player game mode. However, wins are not incremented in the database after a user wins a game, so a user will always have zero wins associated with their username. The lack of complete implementation was due to time constrictions. If given more time, an increment function in the database that is called upon a user win would solve this issue.

5.4.4. Undoing Move in One Player Mode

Another requirement feature that was originally listed was undoing a move. This was implemented in two player mode. However, the automatic call to AI in one player mode would have made it necessary to keep track the last two moves in order to undo. If given more time, the full implementation would be possible through this method.

6. Peer Review Results

The purpose of the peer review is to receive an evaluation and recommendation of milestone one based on the requirements, quality attributes, notation, and design of the original document.

6.1. Recommendations

The following table shows group 10's recommendations, ranked in order of importance.

Table 5. Group 10's Recommendations.

ID	Recommendation	Applied
R1	Either add a restriction only allowing one user to communicate with the server at a time or change the architecture to support multiple users (since currently the system appears to crash if multiple users attempt to access it at the same time).	
R2	Add error-handling (since currently the system appears to crash or have ambiguous behaviour if an error occurs, for example the AI move request times out or the user is logged in at multiple locations).	✓
R3	Server side user validation (if user validation is done on the client side, the user would be able to see everyone's username and password. Serious security concern!	
R4	Users should be able to see their previously won games if they are logged in rather than by screen name (requirement 2.2.8) since the user may not want others to be able to see their score (alternatively, add an option for making the score public or private).	
R5	Consider not using flyweight pattern because it adds overhead.	✓
R6	Improve AI interaction, possibly add an interface (allowing the client-side program to send less structured data to the server since the query is currently being made on the client-side. Also, to support additional functions from the AI like army detector).	✓
R7	Consider more suited design patterns such as singleton for only having one instance of database connection.	✓
R8	Store the salted password hashes only for added security (instead of plain text).	

6.2. Remedies

As recommended by group 10, we added error handling when making requests to the server, database and the AI (R2). This was necessary to prevent the application from crashing entirely. In addition, we chose to not use the flyweight pattern (R5) from our original design (see milestone 2). Instead we used a method of creating tokens individually whose colours are interchangeable based on mouseover or onclick. In addition we chose to add a AI interface (R6) which allows the data being sent to the server to be structured; it also enables proper response handling. Finally we incorporated the singleton pattern for our database connection (R7).

If given more time we would enable multi-user communication with the server (R1). Currently the system can only handle one game request at a time. In addition, we would implement more security features such as R3, R4 and R8. In the time period given these recommendations were not prioritized.

7. Conclusion

The final Go Web Application successful and an excellent learning experience. Prior to this class, our group had little to no experience in software design or web development. Through requirements and design analysis, model driven engineering, and the use of design patterns, we were able to solve problems encountered throughout each milestone and produce a deployable web application. Overall, the final design project was an excellent and exciting introductory lesson into software architecture.