PROJECT Design Documentation

Team Information

- Team name: team e
- Team members
  - Ben Jordan
  - Adam Mercer
  - James Clementi

Executive Summary

Purpose

The purpose of this project is to create a web based checkers application that allows the user, anyone who wants to play checkers, play remotely with another user.

Glossary and Acronyms

Term	Definition
VO	Value Object
GB	Game Board
MVP	See Definition in Section 2

Requirements

This section describes the features of the application.

Our Trello board contains all the requirements needed for a successful implementation of the project. Important features include Player Sign-in, Starting a game, Player Sign-out, Making a move, Promotion, Resignation, and Winning the game. All of these stories pertain to the Gameplay and GUI epics. For example, A promoted piece must be represented graphically and with the ability to move backwards.

Definition of MVP

The MVP for this project involves the ability for two players to be able to log in and out of aWebCheckers lobby, play a game where they make moves abiding by the rules, are able to play until one player has no more pieces or one resigns, and declaring victory for the player with at least one remaining piece.

MVP Features

Notable MVP Features include Player Sign-in, Player Sign-out, Start a Game, Make Move, Resignation, and Has Won, which all fall under the Gameplay or GUI epics.

Roadmap of Enhancements

Some enhancements we plan to include are the ability to Replay a match after it has concluded and Spectate a match as it is going on.

Application Domain

Domain Model:

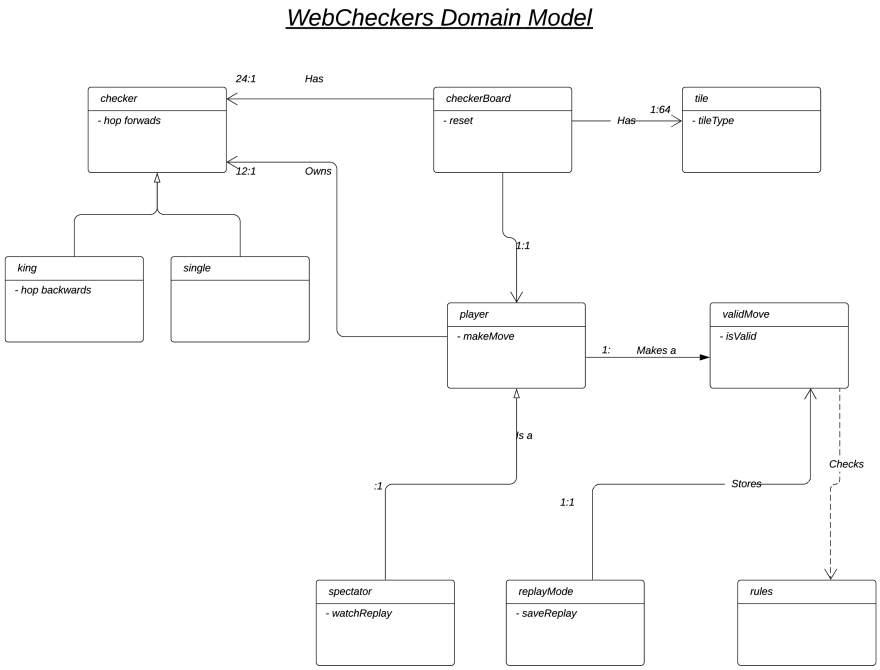

Figure 1: The WebCheckers Domain Model

In our domain model, we've highlighted the key entities that are involved in process of playing a game of checkers. The game begins with a player, who is either in the regular play mode or spectator mode. The model then shows how the player has a single GB and interacts with each of the 24 pieces on the GB by making a valid move.

Architecture and Design

This section describes the application architecture.

Summary

The following Tiers/Layers model shows a high-level view of the webapp's architecture.

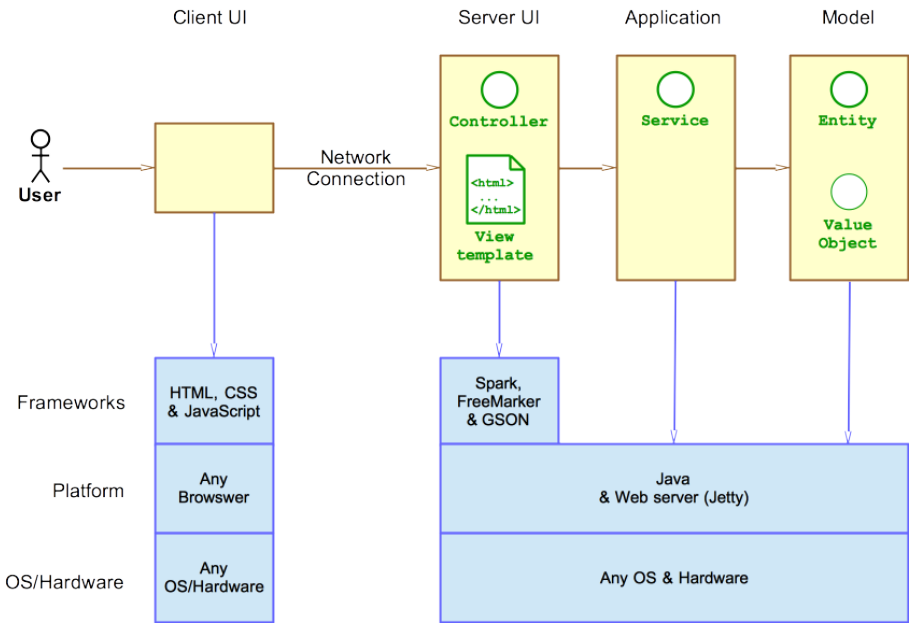

Figure 2: The Tiers & Layers of the Architecture

As a web application, the user interacts with the system using a browser. The client-side of the UI is composed of HTML pages with some minimal CSS for styling the page. There is also some JavaScript that has been provided to the team by the architect.

The server-side tiers include the UI Tier that is composed of UI Controllers and Views. Controllers are built using the Spark framework and View are built using the FreeMarker framework. The Application and Model tiers are built using plain-old Java objects (POJOs).

Details of the components within these tiers are supplied below.

Overview of User Interface

From the user perspective, the ui in this application is fairly straightforward. Once the user enters the website, they are brought to the Home page. This page

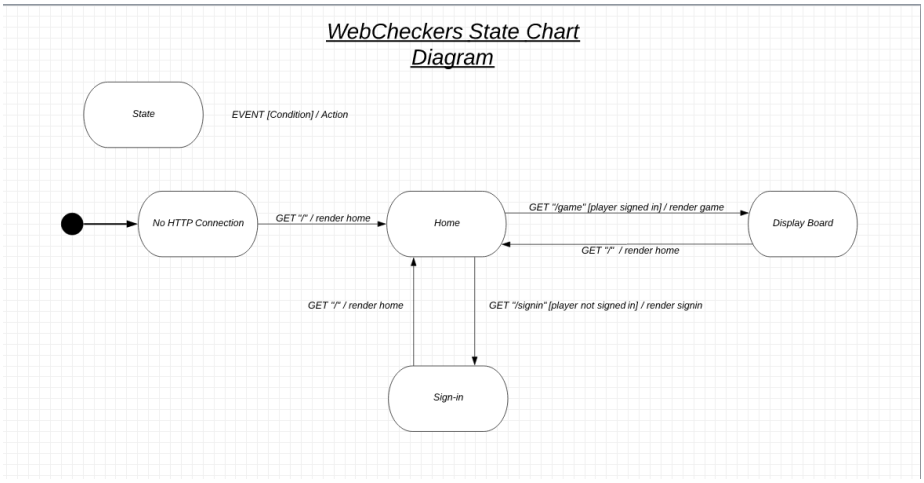

Figure 3: The WebCheckers Web Interface Statechart

allows a user to sign-in, taking them to the Sign-In page solely dedicated to this task. The Home page also allows users to see other users and start a game with them. Doing so brings the user to the Game page.

UI Tier

In our UI tier, we have the route classes. These classes are responsible for handling the initializing and transferring data between the Java components and the Spark components of our project. Each Route class is used depending on what page the user is accessing. Currently, we have a GetGameRoute, GetHomeRoute, GetSigninRoute, PostResignRoute, PostSigninRoute, and PostSignoutRoute. All of these classes have similar signatures, but differ in their functionality of the handle function.

Here's an example of what the GetHomeRoute class looks like.

Inside handle, the main functionality lies in a ModelandView hash table used to transfer data between the spark ui and the Java components, and a template engine used to render the ui itself.

This sequence diagram shows a timeline of what goes on in the handle function in GetGameRoute.

Application Tier

For the application tier of our project, we've decided to use three components, the PlayerLobby, the WebServer, and the GameCenter. Starting with the PlayerLobby, this is a very simple class designed to store the all of the current Player instance, each of which represent an actual user who's online. The

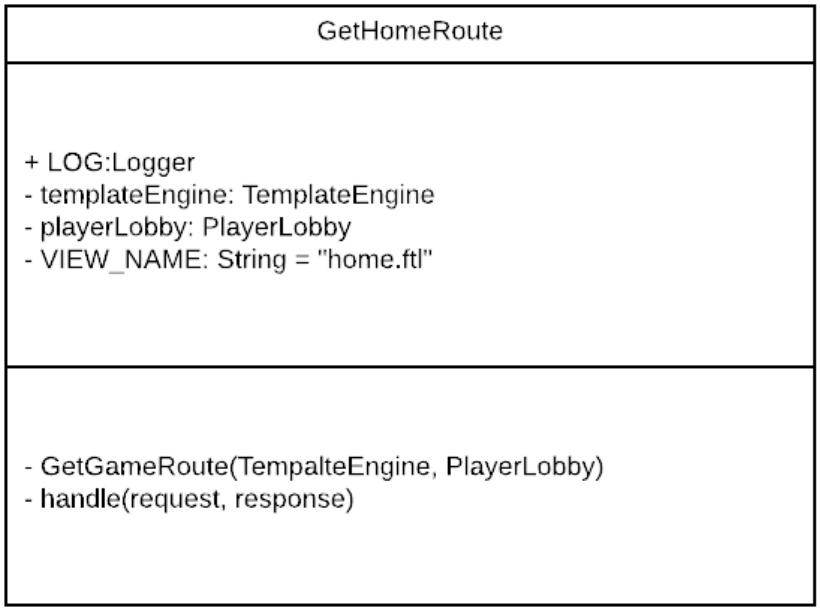

Figure 4: GetHomeRoute

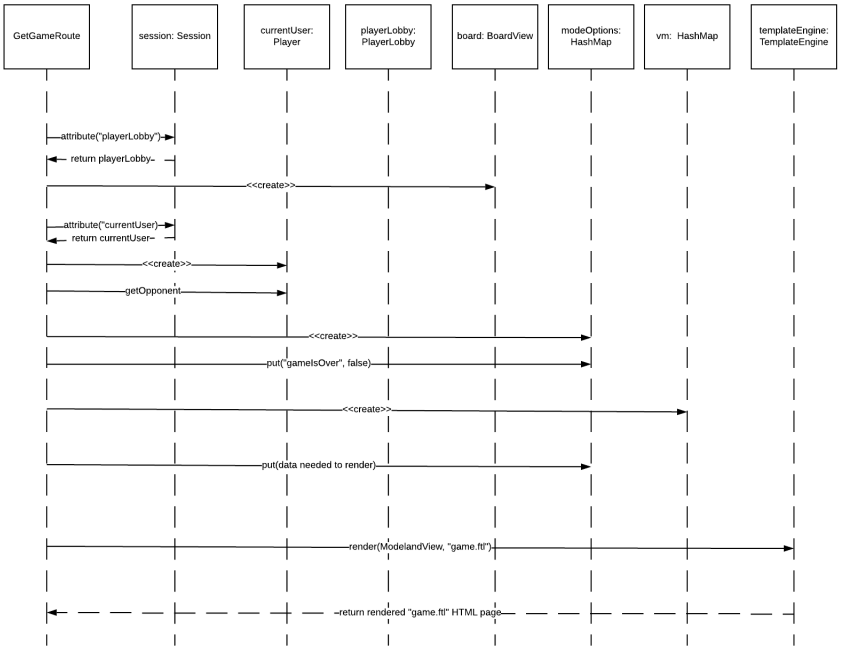

Figure 5: GetGameRoute Sequence Diagram

class stores players as an ArrayList, and its functionality involves querying this arraylist.

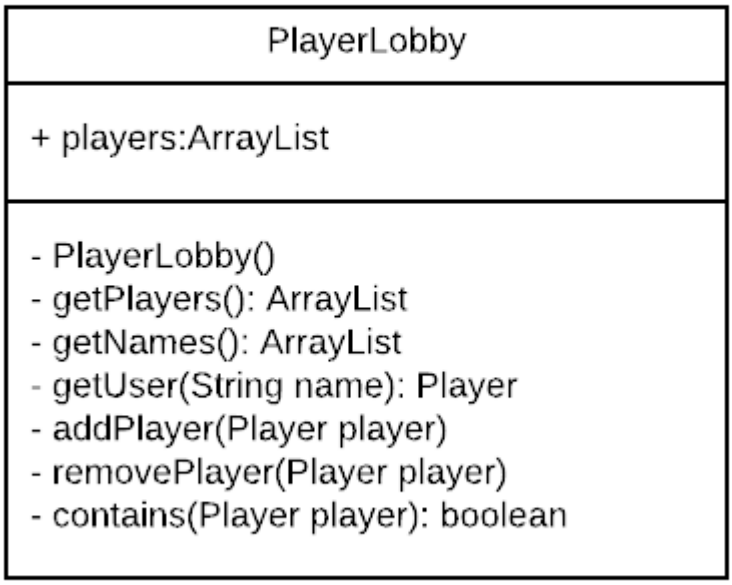

Figure 6: PlayerLobby

As for the other two components, the WebServer is responsible for coordinating the UI Routes. It stores route urls, and helps to initialize the pages when accessed by the user. The GameCenter class is essentially to the BoardView what Player is to PlayerLobby. This class stores game instances and information about these instances. We haven't currently implemented most of GameCenter's functionality, but this is what our future plans our.

Model Tier

The purpose of our model tier is exactly what you might assume. These classes are deigned to model the domain of a real life checkers game. The classes in this tier include BoardView, Piece, Row, Space, and Player. For our design we are planning on creating a Game class to model a single game instance and its rules.

What one might consider to be the main class in this tier is the BoardView. The flow of the model begins with the Piece class, which are created and added to

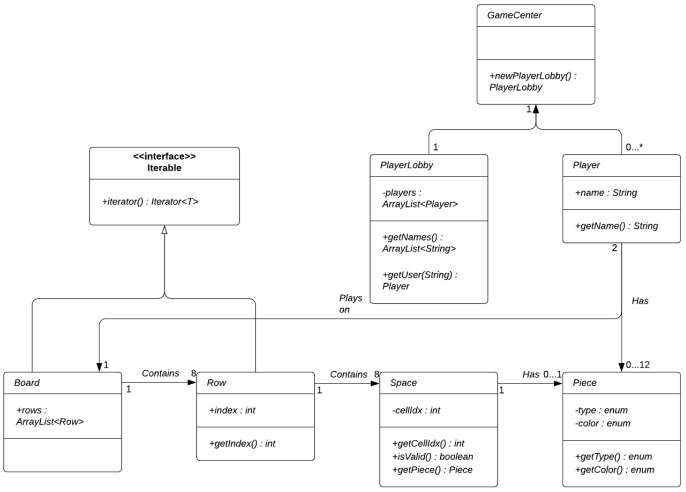

Figure 7: Model Tier UML

instances of Space. These istances are added to Row instances, which are added to an arraylist of Rows in BoardView. As you can see, they all work together to create the BoardView

Design Improvements

Our design could be improved to adhere to the object oriented design and model view controller architecture by adding guard conditions to the signin POST requests. Home should have a loop back transition for refreshing every 10 seconds. Routes should be consistent with how they access the list of players and players should not store their opponents in the player class.

Testing

Our testing strategy involved debugging through a debugging tool, through periodic print statements, and unit tests. We frequently test elements of the story we are working on as we develop it and run more thorough tests after it is "finished". For example while developing the PostSigninRoute, tests were done after each if statement was made to verify the logical conditions of each if statement could be met and the correct result was achieved. After each if statement was created and the class was finished some unit tests were written and manual tests were ran.

Acceptance Testing

Currently, two user stories have passed all acceptance criteria tests (Player Sign-in and Log-Out). Starting a game has some of its acceptance criteria tested, however, not all these tests pass. Most of our stories are not ready for test- ing such as, Resignation, Has Won, and Make move. Most of our issues are from lack of implementation.

Unit Testing and Code Coverage

Unit tests cover only part of each tier, the application tier has tests for the GameCenter class which test 100% of the methods in GameCenter. The Model tier has unit tests for the Piece and Player classes which both have 100% of the code covered. The UI tier tests include GetHomeRouteTest, GetSigninRouteTest and PostSigninRouteTest these classes were covered only partially and need more tests to be complete. Each tier has classes that are yet to be tested and tests will be implemented in the coming sprints.