# WebCheckers Design Documentation

## Team Information

- Team name: Team C/Group Bee
- Team members
- Peter Kos
- Donald Craig
- Danilo Sosa
- Joseph Saltalamacchia
- Elizabeth Sherrock

## Executive Summary

The WebCheckers application is an online version of the classic board game checkers. Users will be able to access the WebCheckers site and sign in with a username. At this point they can see any other players online and whether or not they are available to play. After selecting an available player both users will be put into a checkers game where they play by traditional American checkers rules. Game play ends when a player wins or when either one of the players chooses to resign. Each player is then brought back to the home page where they have the option to challenge another player or they can choose to sign out and leave the web application.

### Purpose

The aim of WebCheckers is to provide people a way to play a game of checkers online with other players from around the globe. The goal is to make this as easy to use and enjoyable as possible for anyone who likes to play checkers.

### Glossary and Acronyms

| Term | Definition |
|---|---|
| AI | Artificial Intelligence |
| CSS | Cascading Style Sheets |
| HTML | Hyper Text Markup Language |
| HTTP | Hyper Text Transfer Protocol |
| MVP | Minimum Viable Product |
| OOP | Object Oriented Programming |
| POJO | Plain Old Java Objects |
| UI | User Interface |
| VO | Value Object |

| Term | Definition |
|---|---|
| debuggability | The act of writing code that is easy to debug. |
| reflection | Inspecting the fields of a class, particularly its private fields. Can be dangerous! |
| controller | A class that serves as a “hub” of functionality for other features. |
| Spark | Specifically Java Spark – the main framework used to structure the project. |
| FreeMarker | The template engine used to render data onto the webpage from a controller. |
| Session | Stores data relevant to a specific user. |

# Requirements

- Player must be able to sign in with a user name.
- The player must be able to sign out if they are signed in.
- Player must be able to challenge another player to a game.
- The Players must be able to play game of checkers online using American rules.
- Either player must be able to resign from a game during their turn if they so choose.
- The players are notified of the winner of the match and are able to exit the game and return to the home screen.

## Definition of MVP

The Minimum Viable Product should be an application such that:
* Every player must sign-in before playing a game, and be able to sign-out when finished playing.
* Two players must be able to play a game of checkers based upon American rules.
* Either player of a game may choose to resign, at any point, which ends the game.

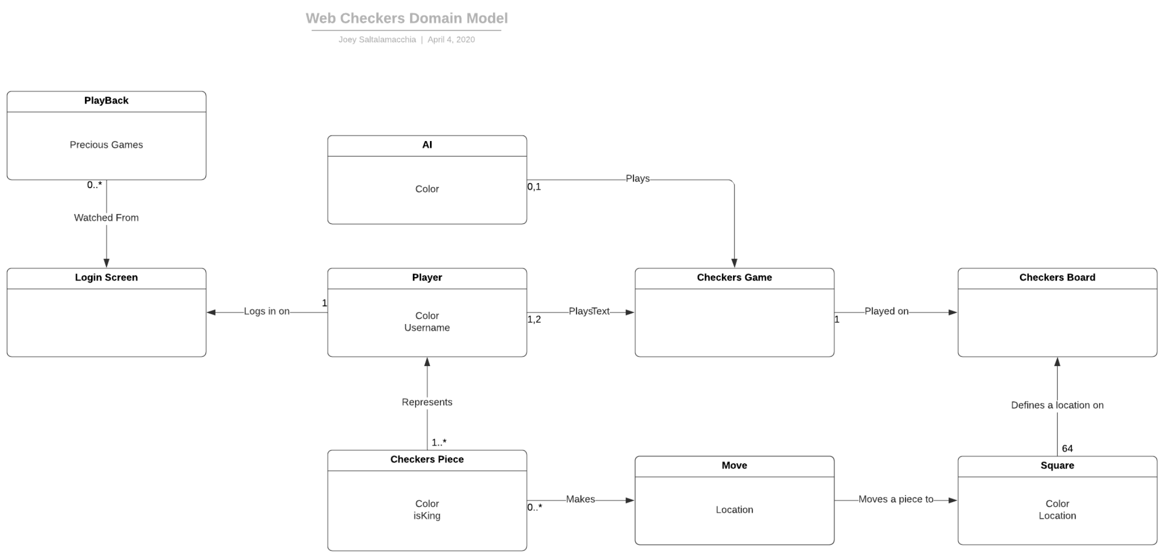
## MVP Features

- Player
  - Sign-in
  - Change player turn
- Play game
  - Start a game
  - Move piece
  - Jump piece
  - King piece
  - Game over

## Roadmap of Enhancements

- Replay mode

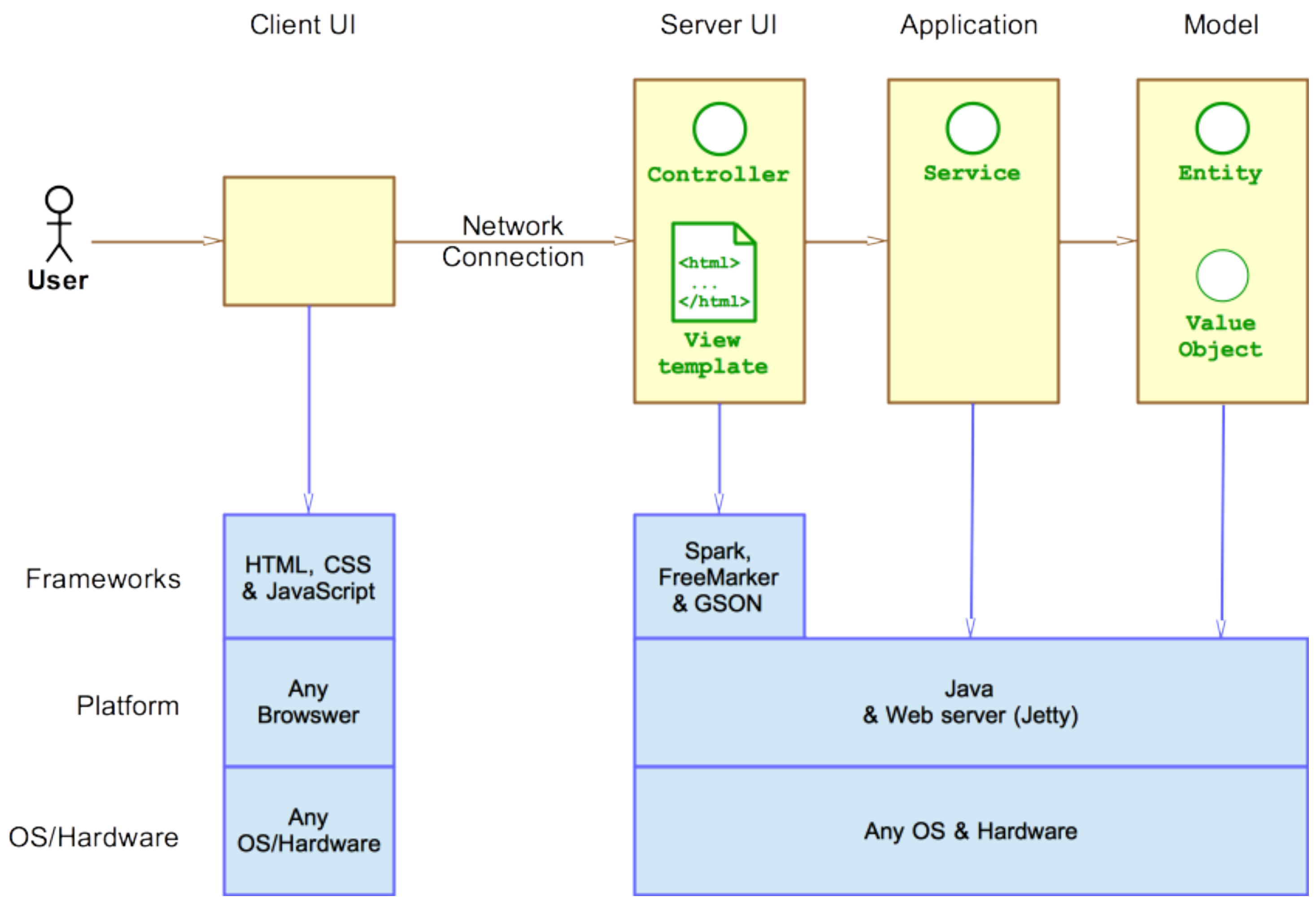
# Application Domain



# Architecture and Design

## Summary

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



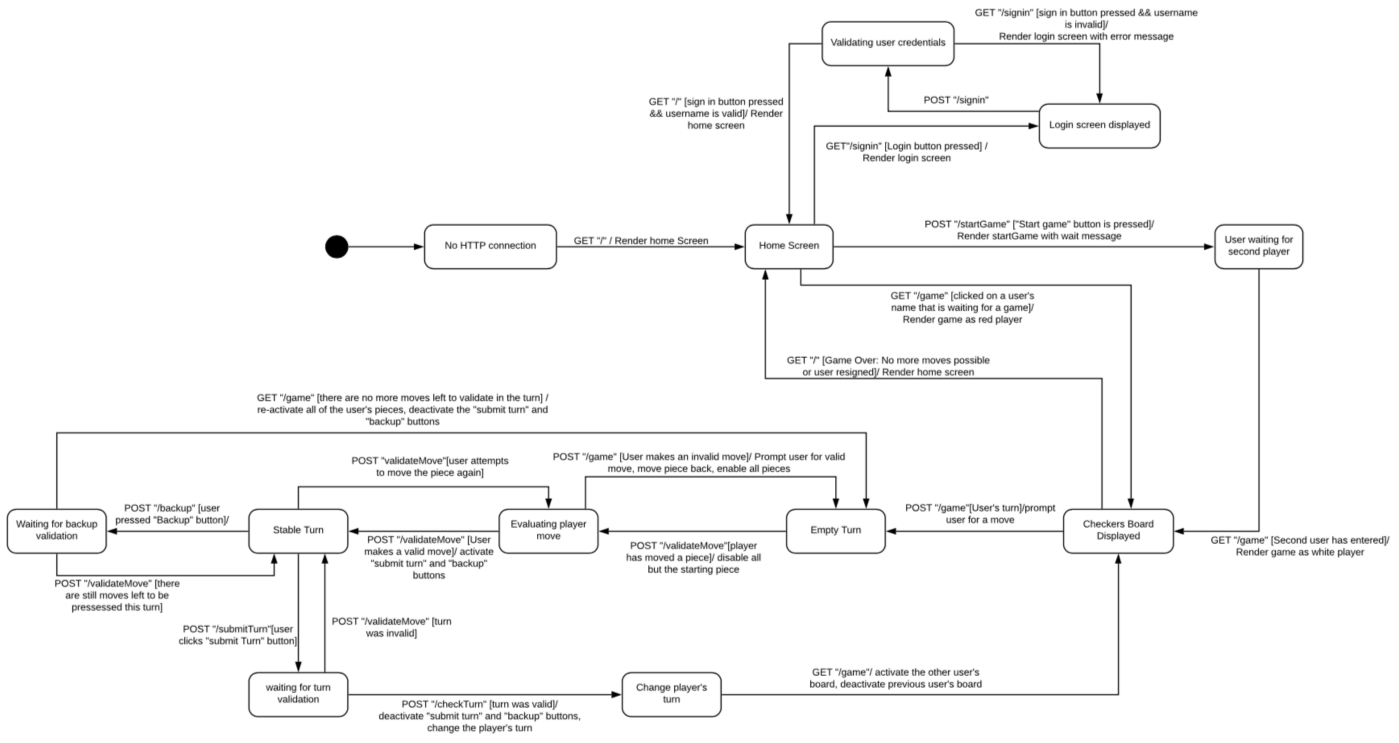
*The Tiers & Layers of the Architecture*

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


Validating user credentials
GET "/signin" [sign in button pressed && username is invalid]/ Render login screen with error message
GET "/" [sign in button pressed && username is valid]/ Render home screen
POST "/signin"
Login screen displayed
GET"/signin" [Login button pressed] / Render login screen
No HTTP connection
GET "/" / Render home Screen
Home Screen
POST "/startGame" ["Start game" button is pressed]/ Render startGame with wait message
User waiting for second player
GET "/game" [clicked on a user's name that is waiting for a game]/ Render game as red player
GET "/" [Game Over: No more moves possible or user resigned]/ Render home screen
GET "/game" [there are no more moves left to validate in the turn] / re-activate all of the user's pieces, deactivate the "submit turn" and "backup" buttons
POST "validateMove"[user attempts to move the piece again]
POST "/game" [User makes an invalid move]/ Prompt user for valid move, move piece back, enable all pieces
Waiting for backup validation
POST "/backup" [user pressed "Backup" button]/
Stable Turn
Evaluating player move
Empty Turn
POST "/game"[User's turn]/prompt user for a move
Checkers Board Displayed
GET "/game" [Second user has entered]/ Render game as white player
POST "/validateMove" [User makes a valid move]/ activate "submit turn" and "backup" buttons
POST "/validateMove"[player has moved a piece]/ disable all but the starting piece
POST "/validateMove" [there are still moves left to be pressessed this turn]
POST "/validateMove" [turn was invalid]
POST "/submitTurn"[user clicks "submit Turn" button]
waiting for turn validation
POST "/checkTurn" [turn was valid]/ deactivate "submit turn" and "backup" buttons, change the player's turn
Change player's turn
GET "/game"/ activate the other user's board, deactivate previous user's board

## UI Tier

### GetHomeRoute

Our architecture begins with the `GetHomeRoute`. It serves to:
1) Instantiate the Session for the user
2) Present the homepage to the user, in various states

The `Session` object contains important information about the current user. In particular, it encapsulates the `UserSession` object, which does the actual storage.

This object is used in other routes, and if any route finds the `UserSession` (or `Session`) object to be `null`, it will redirect to the home (`/`) route to instantiate it. This way, by the time we reach a “later” controller (i.e., signin route), we are sure that the operations in `GetHomeRoute` are complete.

> *A notable exception of this is in the `GetGameRoute`.*

Now, what are these operations?

First, the `Player` is grabbed from the UserSession. The `title` and `NumberOfUsers` is set for the view model, to be presented on the screen.

1. If the player is null, the default (blank) homepage is shown.
2. If not null, and the user is in a game, they will be redirected to `/game`.
3. If not null, and the user is not in a game, they will get a custom welcome message with their username, and see a list of other signed in players.

## GetSignInRoute

Signing in is the next logical step for a new user. Or rather, the *only* valid step.

It redirects to home (`/`) if the `UserSession` is not defined. Otherwise it shows the sign in screen.

## PostSignInRoute

Now, once the user is signed in, the `PlayerLobby` checks if the username is valid.

> *Valid usernames are usernames that contain at least one alphanumeric character, and is not already in use.*

At this point, the username is valid. A `Player` is created, added to the session, and the user is redirected to home (`/`).

## PostStartGameRoute

After the redirect to home, if there are other players online, they will be displayed to the user, who is able to select an opponent. The home page contains an HTML form which is used for the selection. Upon submission, the page submits a `POST` request which simultaneously redirects to `/startgame`.

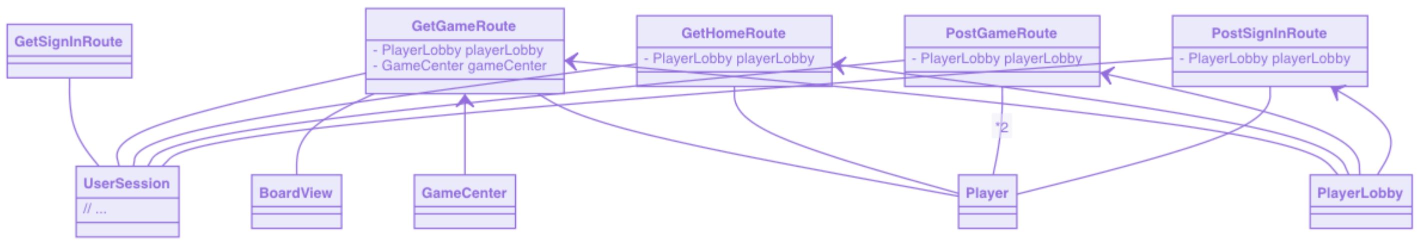
`/startgame` is the middleman between the home page and the game page, checking that the conditions to start a game are met. The `POST` request contains the username of the desired opponent that was submitted via HTML form. The route takes that username and uses it to retrieve the `Player` from `PlayerLobby`. If the second player is `null`, the user is redirected to the home page. Currently players can only participate in one game at a time, so if the second player is already in a game, the user is also redirected to the home page, this time with an error message explaining that the desired player is already in a match. If neither of those conditions are satisfied, the first player (the user) is assigned red pieces, the second player is assigned white pieces, and after being marked as playing a game, both are redirected to `/game`.

## GetGameRoute

Let the game begin!

This page utilizes classes in the Model Tier and displays the board to the players. The Model Tier components will be explained in a separate section.

## UI Tier diagram



If this diagram reveals a complicated and moderately coupled structure, this is normal. Our controllers make heavy use of a *small* subset of "abstraction" classes – `BoardView`, `UserSession`, `PlayerLobby` are the strongest contenders. (Admittedly, `Player` could be factored out.) Additionally, not shown here is the all-important `Constants` file, which almost every single class is coupled to. This is expected and encouraged in our design as it:
1) Increases readability of commonly used strings,
2) Removes ambiguity about where commonly used strings should go, and
3) Takes advantage of functions to make "format" strings easy to read in any context via IDE-autocomplete.

# Application Tier

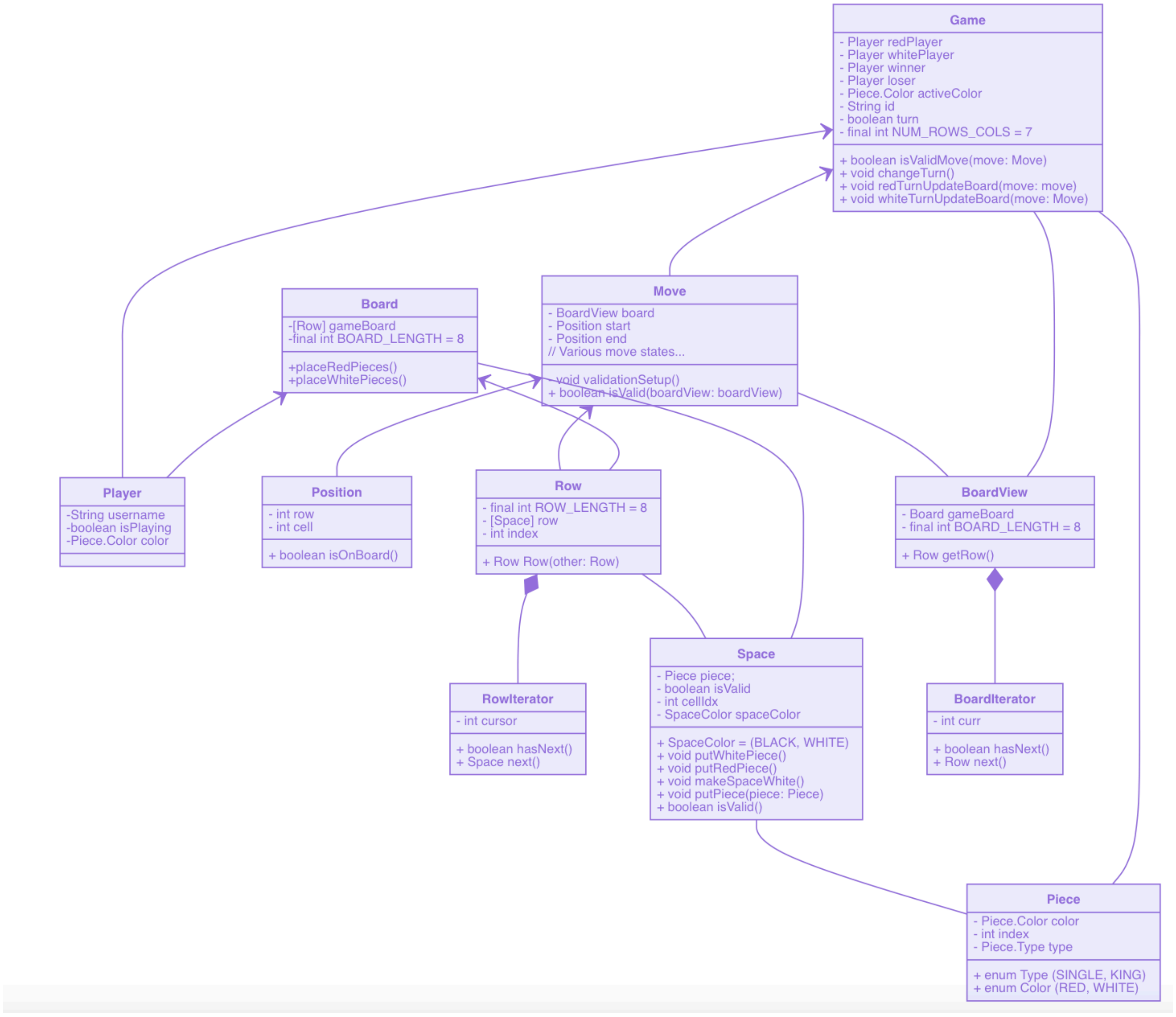
The Application Tier handles application-wide information that is important for both the users and developers. These are the classes that make up the Application-Tier:

- GameCenter - Contains a HashMap of games in which the game Id is the key and the Game object is the value. With this class we can add new games as they start, get a game by searching for the Id and check if there are existing games. None of these data is pertinent to the user so it is not shown in the View.
- PlayerLobby - Contains a HashMap of players in which their username is the key and the Player object is the value. In this class we can add and remove players as they sign in and sign out. We can also get the complete list of players and the amount of players. We use a list of players without the current player to show what players are available to play. Finally, if a user enters WebCheckers without login we will only show the amount of players online.

## Model Tier

The Model Tier handles most of the application logic and consists of the following classes that make up the major components of the WebCheckers application:

- Board – Consists of an array of Rows. Represents the board in our project, including all board-modifying options.
- BoardView - Serves as an abstraction for any views. Primary goal is to present a properly-oriented board for each player, such that their color is on the bottom of the screen.
- Game – Contains the game entity consisting of two players and the BoardView.
- Move – Defines a move in the game. This model takes care of ensuring a move is valid before it is accepted and the game board is updated.
- Piece – A checkers game piece, can be either Red or White.
- Player – The player entity that stores the unique username, player status and color.
- Position – Represents a location on the checkers board defined by the row and cell position on that row.
- Row – The game board consists of eight rows, each of which is an array of eight spaces.
- Space – Represents an individual location on a row. Each space keeps track of it’s color, whether or not a piece is on it and if it is a valid place for a move to be made.



## Design Improvements

### Constants Refactoring

The first design improvement is ongoing, which is the **refactoring** of String constants into a singular file.
This is incredibly useful for two reasons: readability and testing.
In the original given code, we were expected to use `String.format` on existing format-specified Strings. Some of these were public, some were private. Some were used only in one file, while others were used widely. This was difficult to read, and made it hard to know when a String “belonged” to one class if multiple classes used the String equally.

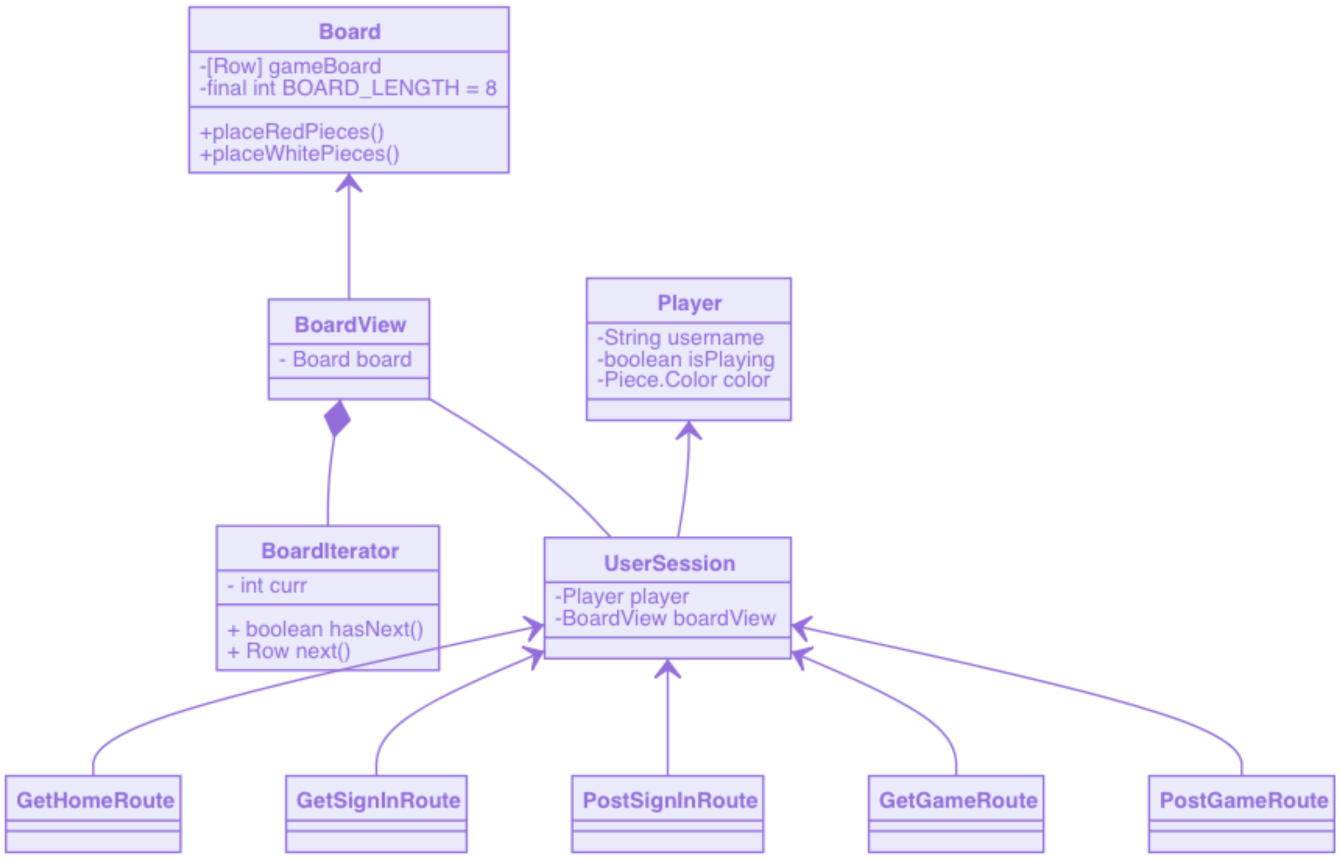
We have since begun writing a `Constants` file that acts as a single source of truth for all String values in the project. This makes the code extremely easy to read (i.e., `Constants.userAlreadyInGameMessage`), and tests can access all of these values for assertions with a single `import`.

As the project progresses, we plan to factor out `enums` such as `Color` to this constants file, as well as other `Messages` and other single-use Strings.

## Note on BoardView

Initially, in sprint 1, our `BoardView` class was improperly holding all of our model data. This required a deep refactor to separate our `Board` model data into its own `Board` class.

Additionally, this gave us the advantage of abstraction between our UI and Model tiers, which is clearly illustrated in the following diagram:



Here, **all** controllers are routed through `UserSession`, to `BoardView`, *before* interacting with any models. This is an important separation of concerns and is a core part of our architecture – as of Sprint 2, at least.

# Testing

## Acceptance Testing

Both of the user stories completed in sprint 1 have passed all of their acceptance criteria. Currently twelve other stories have not had any testing yet, with half being in the sprint backlog and the other half in the product backlog.

We strove to maintain a strict style where people who wrote code for a card did not also test that card. This was easier in the beginning because we worked more synchronously, and it was easy to tell who had coded and not. However, as we all started to contribute to a single card, nobody was left who could test.

Instead, we let anyone test who wanted to. This made it easier for team members to contribute to each card rather than keep an entire card to themselves.

During acceptance testing of the player sign-in user story, there were certain points that failed in the first round of testing:

### Start a Game

The only setback during the Start a Game testing was the view of the pieces in the board. At first the pieces didn't show at all in the board, but after fixing the `isValid()` method in the Space class pieces were now showing. However, they were not showing in the correct positions. Finally, we had to fix the logic in an `If` statement from the `Row` class in which the assigning of valid spaces depending of the index was wrong.

### Sign in card

This card was also responsible for the removal of a procedure: to re-run through all acceptance criteria when a single one is invalid, to prevent regressions.

However the sign in card hit us with many, many bugs, up until thirty minutes before our initial demo. Enough **refactoring** led us to a working solution, but not without much confusion. We settled on making our unit tests more thorough since this card.

## Unit Testing and Code Coverage

Our unit tests followed three main principles:
1) Be self-contained
2) Be thorough
3) Be descriptive

To be self-contained, we tried to rely as little as possible on other tests, or other classes, creating mocks where needed. The only exception to this was getter methods.

> *We assume that getter methods are fully functional to avoid a need to use reflection, which is error-prone at best and mysterious at worst.*

In descriptiveness, we sought to make our method names as verbose as possible.

For example, in `PostSignInRoute`:

```
public void testUserNameIsValid() { ... }
```

versus

```
public void testWhenUsernameHasNoAlphanumericCharactersShowInvalidMessage() { ... }
```

This is due to the principle of **debugability**.

If someone broke some code, the error message is the test name itself, as then the programmer doesn't have to debug the test.
In this case, if the above test did not work, the user would know that the single use case of a username with *no* alphanumeric characters would *not* be showing an invalid message, when it should.

Two paths of logic are immediately available:
1. The username with *no* alphanumeric characters is broken somehow
2. The invalid message is not set properly in code

If we peek into the component test code, we can see this logic represented *directly* in the source:

`PostSignInRoute.java`

```
// Check if the username is valid: that is,
// a username that contains at least one alphanumeric character,
// and is NOT already in use.
// If not valid, show an error to the user.
if (!playerLobby.isValidUsername(username)) {
    // ...
    viewModel.put("message", Constants.invalidUsernameMessage);
    return templateEngine.render(new ModelAndView(viewModel, "signin.ftl"));
}
```

This direct mapping between test architecture and code structure is a key element of our test design.