Web_Checkers Design Documentation

Team Information

- Team name: CheckChamps
- Team members
- Cole Hornbeck
- Jenna MacBeth
- King David Lawrence
- Jakob Rossi

Executive Summary

Creating a checkers game so that users can play against each other.

Relative Figures are contained within the last pages because MarkDown is bad at formatting images and text together.

Purpose

The goal of this project is to recreate the game checkers for a user to play. Players should sign in to a webpage and be able to choose from a list of other players whom to play against. They will then play a game of checkers, following the American rules.

Glossary and Acronyms

Term	Definition
VO	Value Object
MVP	Minimal Viable Product
AI	Artificial Intelligence
UI	User Interface
Model	The system layer in which data and logic is stored
Application	The system layer in which data is manipulated to be used by the UI

Requirements

- Make it so that players can play against each other in a game
- Make it so that players can move their piece
- Make it so the user can sign in
- Make it so the user can choose an opponent to play against

Definition of MVP

The MVP (Minimum Viable Product) for this project is for the user to be able to easily and successfully play a game of checkers against one opponent. The player should also be able to sign in, sign out, start a game, and resign that game if so inclined.

MVP Features

- Sign In
- Sign In With Account
- Choose Opponent
- Play Game
- Resign Game
- End Game
- Sign Out

Application Domain

The game implements basic ideas of a game of checkers

See Figure 2 for Project Doman Analysis

Architecture

This section describes the application architecture.

Summary

The Model tier contains the low-level pieces of data needed for the game (Piece, Board, Player). The Application tier takes this data and gives it to the UI so that it is interpreted and made usable for the user.

Overview of User Interface Tier

The UI includes the routes for multiple pages so that the view templates can be generated. This includes the GetGameRoute, GetHomeRoute, GetSigninRoute and GetSignoutRoute classes. The UI also lets the user input data. The PostSigninRoute and PostGamestartRoute classes allow this. The general form of the UI calls are shown in the image below.

See Figure 3 for Overview of UI calls and functions

GetGameRoute

This route generates the game redPlayerBoard from given information. It also sets up the playerLobby and players in it so that opponent also gets kicked into the game.

GetHomeRoute

This route generates the home page. Depending on if the user is signed in, also shows the other signed in players.

GetSigninRoute

This route generates the sign-in page for the user. It allows the user to sign in and is the first step to playing a game.

GetSigninAccountRoute

This route generates the login page for signing in with an account. An account will persist over log-in and log-out.

GetSignoutRoute

This route is called when the user clicks signout. It generates no webpage, but instead manipulates the values inside the backend to sign the user out and return to the home page.

PostGamestartRoute

This route is called when the player clicks on another players name. It starts the initialization of the game and redirects to GetGameRoute

PostSigninRoute

This route accepts the input from the user when they enter their name in the signin screen. It uses this name to create the user instance and allow the player to play games.

PostSigninAccountRoute

This is the route that receives and processes a username. It also does the functionality of adding an account to the PlayerLobby and setting the account player into the lobby like normal.

PostValidateMoveRoute

This is the route that processes and validates whether or not a move can be made. The flow through this route contains the majority of board modifications. During the call of this route, the game board is updated for the most recent move made.

PostCheckTurnRoute

This route gets called in regular intervals to determine whether the active player has submitted or resigned their turn yet.

PostResignRoute

This is the route called when a player wants to resign the game. It removes the player from the game and sets the necessary variables for ending the game.

PostBackupTurnRoute

This is the route called to undo a move. It has only a few lines of logic, but succeeds in removing the most recent move made on the board.

PostSubmitTurnRoute

This is the route that handles submitting and changing the turn. It changes all necessary states to the turn of the other player, and kicks both players through getGame again to refresh the boards, at different points.

WebServer

This piece initializes all of the routes and allows them to be called by their respective URL paths.

Overview of Application Tier

The application tier includes the game controller class, PlayerLobby. This contains information that is stored in the system and gives it to the UI-Tier classes. More specifically, Player information is stored, and players can be added and/or removed from the lobby. It also provides information about the game and has the ability to put players into a game.

Overview of Model Tier

The model tier contains classes that represent each component needed for a checkers game to take place.

BoardView

This class generates the redPlayerBoard so that a game can take place, and is made of rows.

Game

Represents a single checkers game between two players, holds information about the entire game. Also performs the major functionality on the game board in the form of moving, processing, and validating moves.

Message

Contains piece of text depending on the action that caused it. Used to create errors and general user messages.

Piece

Represents a single piece that a certain player uses based on the color of the player (red or white).

Row

Represents a row of spaces on the redPlayerBoard, it is used by the BoardView class.

Space

Represents a playable space on the redPlayerBoard that a piece can be placed on.

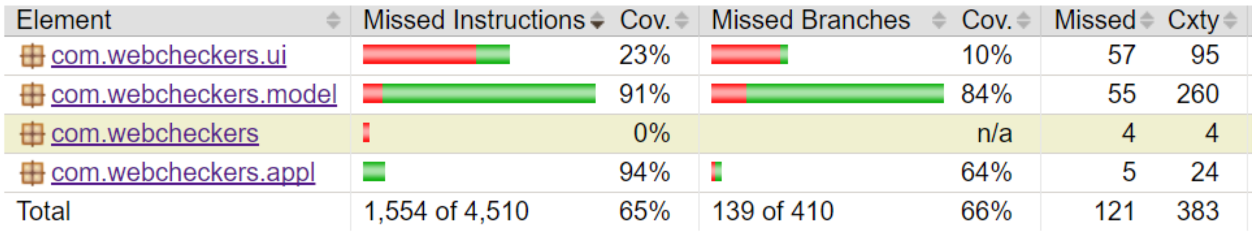
Static models

See Figure 4

Dynamic models

See Figure 5

Code Coverage



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty
com.webcheckers.ui		23%		10%	57	95
com.webcheckers.model		91%		84%	55	260
com.webcheckers		0%		n/a	4	4
com.webcheckers.appl		94%		64%	5	24
Total	1,554 of 4,510	65%	139 of 410	66%	121	383

Figure 1: Top-Level Code Coverage Image

This image shows the code coverage percentages of the project so far. The Model tier is the most well-tested area, mostly just due to the fact that its classes are the easiest to test. The UI tier has far less testing coverage due to the fact that the classes in it are more difficult to test. Because they generate an image, their return values are not a hard value that can be easily tested like Model. The Application Tier, which as of right now only consists of one class, has not been tested yet.

Code Metrics

Complexity

For the most part, the only classes that went above the complexity threshold were UI classes. The only non-UI classes that didn't meet complexity requirements were Game and BoardView. This is due to too many responsibilities being held in one single class.

Martin Package Metrics

Abstractness: 0

There were no abstract classes throughout the entire project. ####Afferent Couplings: 11

Efferent Couplings: 20.25

Distance from Main Sequence: 0.33

This means that the distance from the desired ratio between abstractness and instability is close to 0, which is desired. This means that the project is not too abstract and not too unstable. ####Instability: 0.70 This means that making changes to a certain package will most likely impact other packages. Since the measurement is close to 1, this is not desired as it negatively influences re-usability.

Lines of Code

Total lines of code: 6,696

Javadocs Coverage

The total Javadocs coverage of the entire project is 15.68%. This means that only that percentage of methods and classes had Javadoc representation that explained its function.

Future Improvements

- Refactoring code from the GetGameRoute class into the Game class, therefore easing the strain on the route.
- Hiding important business-class logic such as the lists and BoardView variables we used.
- Reducing code that was used in previous iterations but is now unnecessary.
- Adding more Javadocs to the methods and classes that need them.
- Refactoring dependencies so that packages are more independent (decreasing instability).

Figures

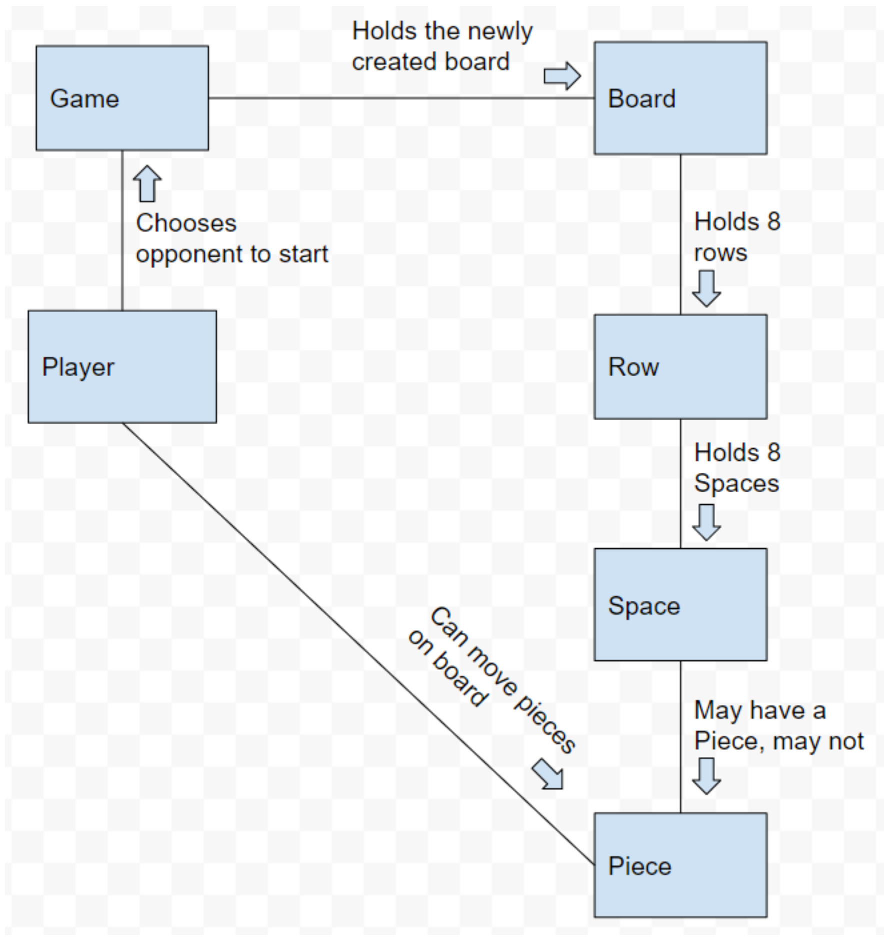

Holds the newly created board
Game
Board
Chooses opponent to start
Holds 8 rows
Player
Row
Holds 8 Spaces
Space
Can move pieces on board
May have a Piece, may not
Piece

Figure 2: Domain Analysis of the WebCheckers Project

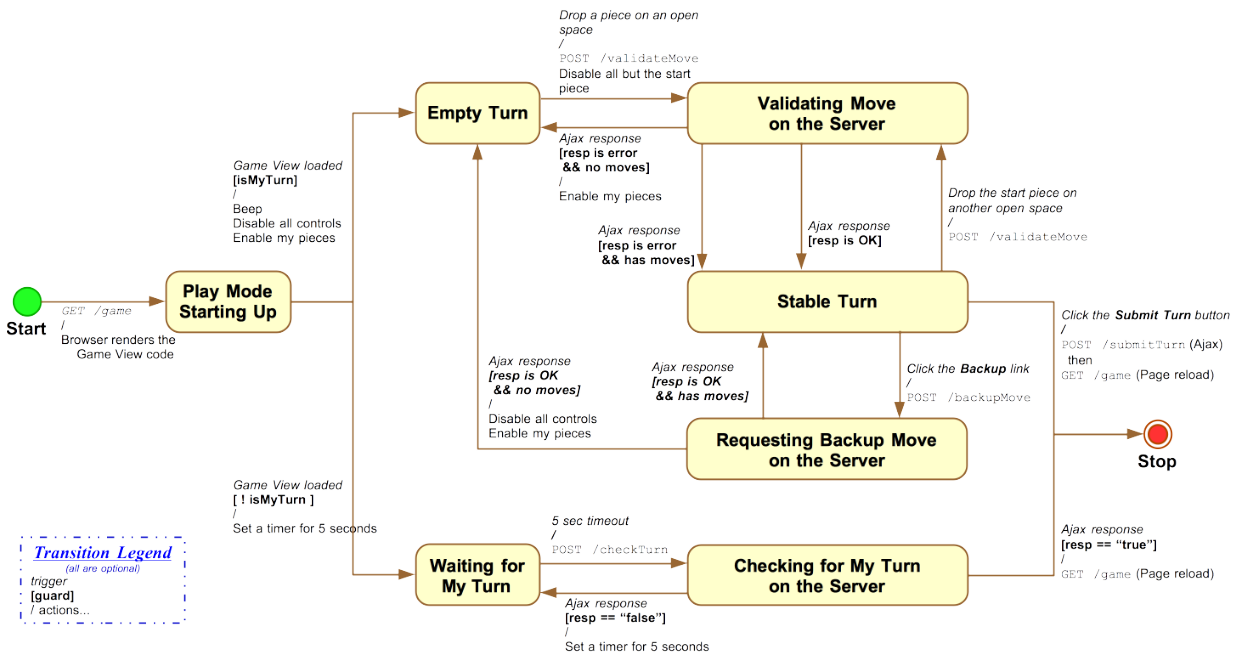


Figure 3: General overview of UI calls and function

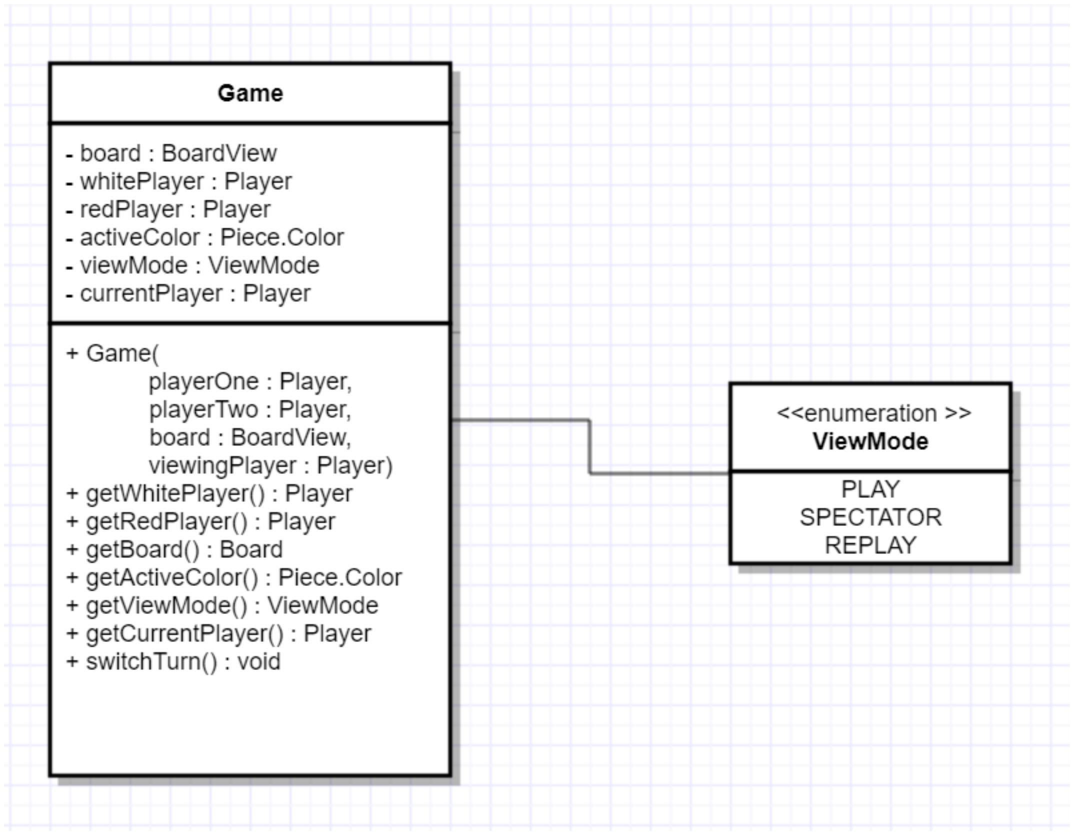


Figure 4: The UML class diagram for the Game class

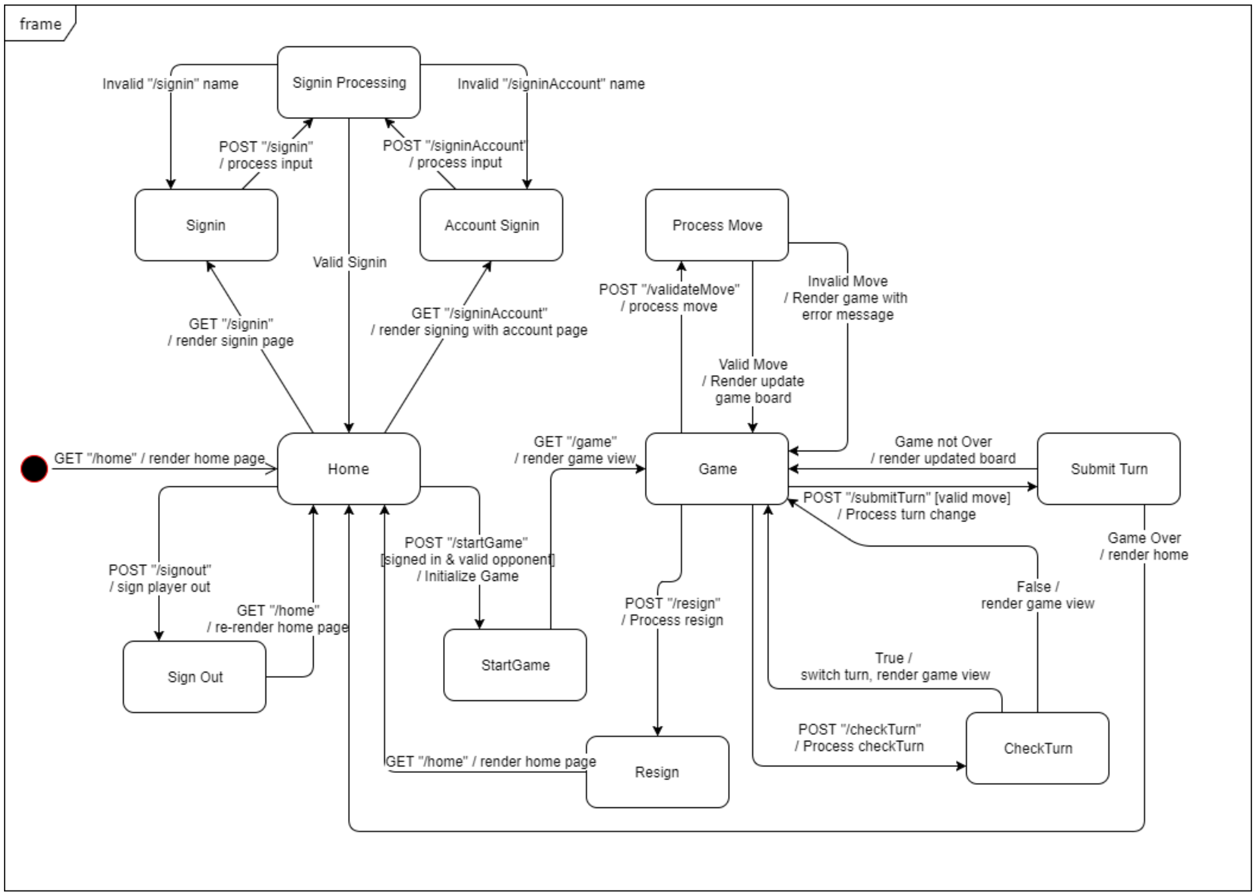

frame
Signin Processing
Invalid "/signin" name
Invalid "/signinAccount" name
POST "/signin" / process input
POST "/signinAccount" / process input
Signin
Account Signin
Valid Signin
GET "/signin" / render signin page
GET "/signinAccount" / render signing with account page
GET "/home" / render home page
Home
POST "/signout" / sign player out
GET "/home" / re-render home page
Sign Out
POST "/startGame" [signed in & valid opponent] / Initialize Game
StartGame
GET "/game" / render game view
Process Move
POST "/validateMove" / process move
Invalid Move / Render game with error message
Valid Move / Render update game board
Game
Game not Over / render updated board
Submit Turn
POST "/submitTurn" [valid move] / Process turn change
Game Over / render home
False / render game view
True / switch turn, render game view
POST "/resign" / Process resign
Resign
GET "/home" / render home page
POST "/checkTurn" / Process checkTurn
CheckTurn

Figure 5: The state diagram for the WebCheckers application

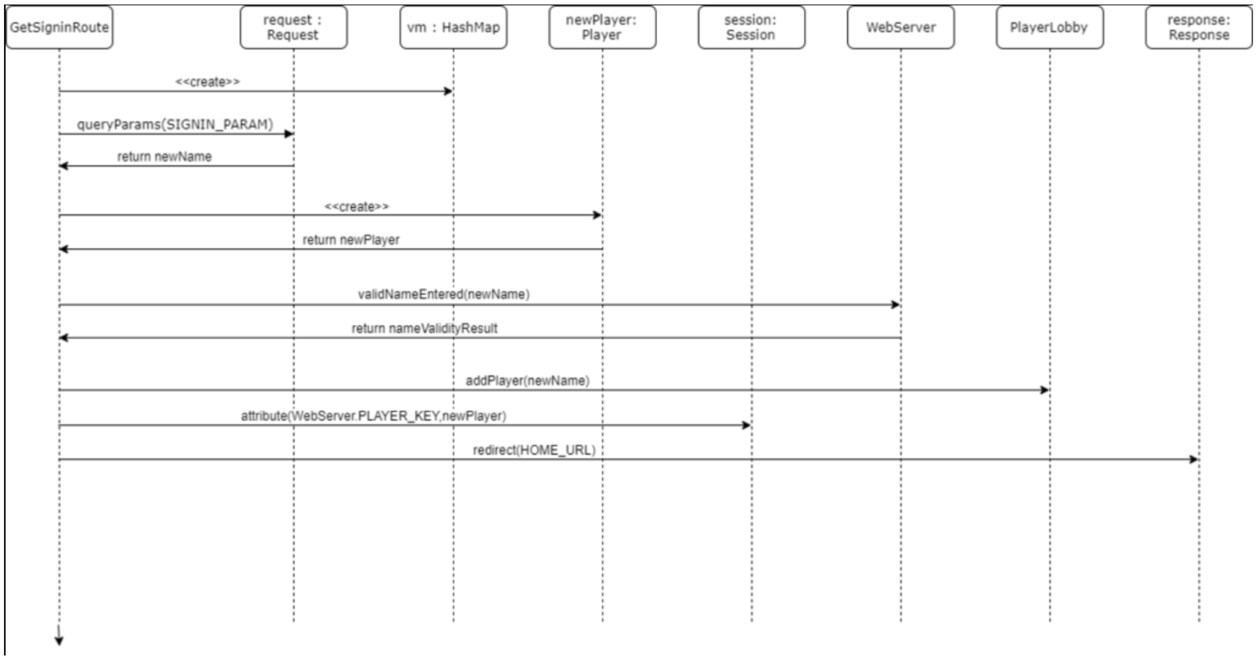


Figure 6: Sequence diagram for Signing in with a valid name