Webcheckers Design Documentation
discountOstrich

Modification Log

Date	Version	Description	Author(s)
2019-10-23	1.0	Initial Draft	Utkarsh Dayal (UD), David Kuehnert(DK), Azhur Viano(AV), Noah Wallace(NW), Nick Atkinson(NA)
2019-10-31	1.5	Revised Initial Draft	AV
2019-11-04	2.0	Finalized for Sprint 2	NW, DK
2019-12-03	3.1	Beginning on Sprint 4 Changes	NW, AV, NA, DK
2019-12-03	4.0	Finalized	NW, UD

Team Information

Team Name: discountOstrich

Team Members:

Nick Atkinson
Azhur Viano
Noah Wallace
David Kuehnert
Utkarsh Dayal

Executive Summary:

This document explains the goal and design of the WebCheckers application. In addition to laying out the MVP requirements, it also outlines the high-level design of the application, justifies design decisions, and states code metrics and test coverage.

Purpose:

This project, *WebCheckers,* is designed to help students develop skills and techniques needed for group software development. Students will implement topics discussed in class in order to complete this project. Such topics include the use of sprints, object orientated design, group organization applications (slack, trello, etc.), code coverage and much more.

Glossary and Acronyms:

Term	Definition
UI	User Interface
MVP	Minimum Viable Product
EBS	Empty Black Square
LSHMSFOAIDMT	Laughing so hard my sombrero fell off and I dropped my taco.
NANDU	A discount Ostrich

Requirements:

1. **A user must be able to navigate to and sign in to the Web Checkers app.**
    - **A player must sign in with an alias that is not currently being used.**
    - **A player has the option to sign-out at any time.**

2. **A player must be able to select another player to start a game.**
    - **The chosen player must have the choice to reject the game.**
    - **If the chosen player is already in a game, a new game will not start.**

3. **On a player's turn they must have the ability to do the following:**
    - **Make a move and submit the move using the control button.**
    - **Make a move and 'backup' to cancel the move before submitting.**
    - **Resign from the current game, causing a GameEnd state.**
    - **Sign-out from the navigation bar at the top of the page.**

4. **A player should only be able to make valid movements, including:**
    - **Making a standard move: piece to EBS.**
    - **Making a capture move: piece over opponent piece to EBS.**
        - **If a capture move is available, standard moves are not allowed.**
    - **Making a multi-capture move: Consecutive capture move(s) with the same piece made after an initial capture move.**

5. **When a player's piece reaches the opponent's King Row - king that piece.**
   - **Updates the piece such that it can move in any diagonal direction.**

6. **When a player eliminates their opponent's last piece, the game ends.**
7. **A player must also be able to select an AI opponent to start a game with.**
   - **The player has a choice between a basic AI and a more advanced AI.**
   - **The AI makes moves after the player allowing for continuous play.**

8. **A player must also be able to observe a game in progress.**
   - **The game in progress will update periodically for the spectator.**
   - **The spectator may stop spectating at any point they desire.**
   - **The spectator may spectate games between players or against an AI.**
   - **When the game ends, the spectator is returned to the home screen.**

Definition of MVP:

A MVP or Minimum Viable Product is a design technique in which, when creating a program, you first create the bare minimum of the project. This means that you code only what is required of the base program, and no extras. In our WebCheckers, this means that we are created the base game of checkers before creating extra features like 'Help' or 'AI Player'

MVP Features:

1. **Every player must sign-in before playing a game, and be able to sign-out when finished playing.**
2. **Two players must be able to play a game of checkers based upon the American rules.**
3. **Either player of a game may choose to resign, at any point, which ends the game.**

Enhancements:

AI Player
This enhancement allows the player to play against a computer player rather than a human. The computer player is designed to have a reasonable level of intelligence to provide challenge to human players.

Spectator Mode
This enhancement allows users to spectate ongoing matches between other players.

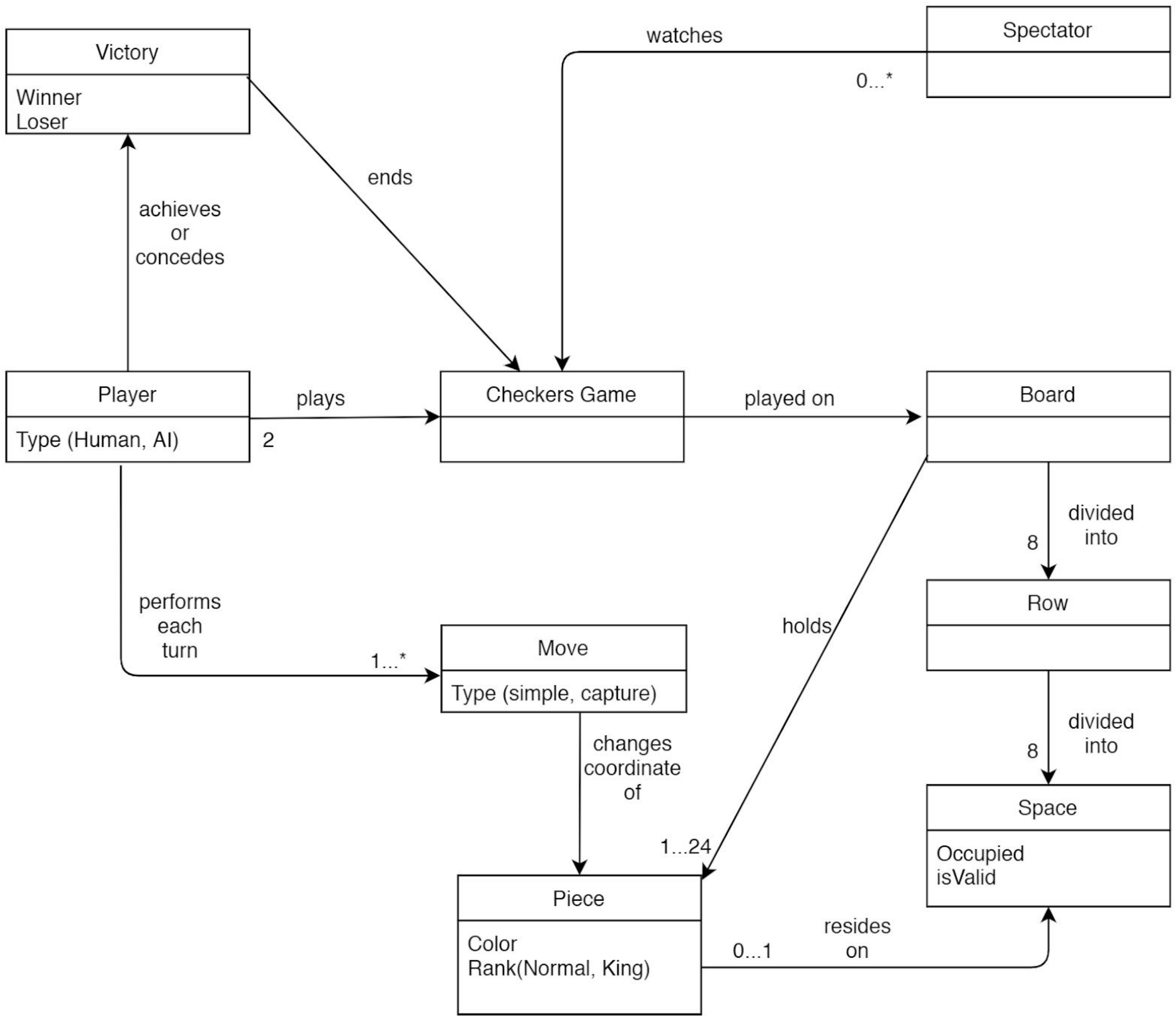
Game Invitation
This enhancement allows users to reject a prospective game invitation, or accept a pending one.

Application Domain:

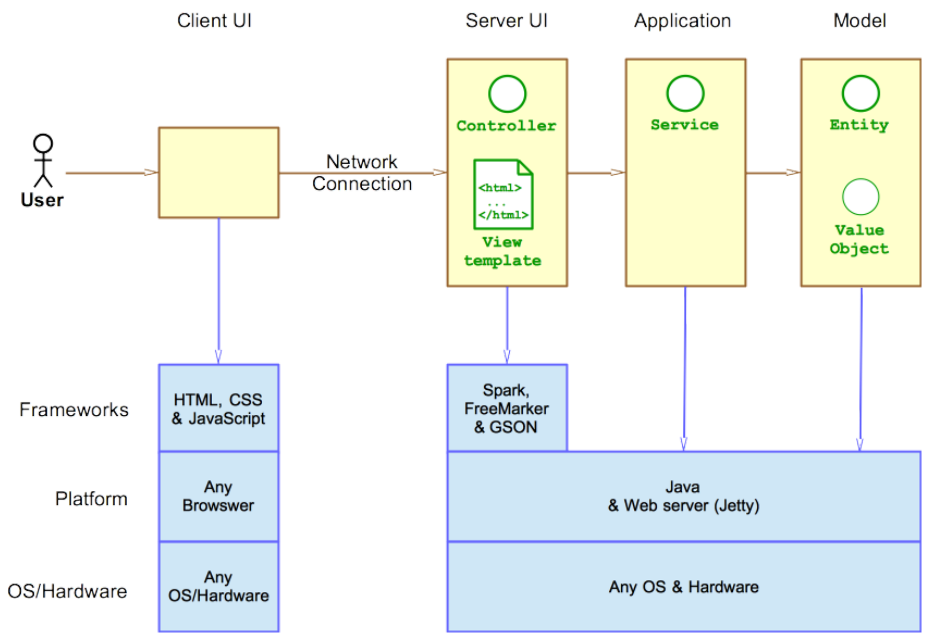
Overview of Major Domain Areas

The domain of the project centers around the Checkers Game, as seen in the diagram below. Any number of Checkers Games may be created, however, it must be played on a board by two players, one of whom can be AI-Controlled. The Checkers Game is played on a CheckerBoard of 64 squares, with a total of 24 pieces (12 per player) residing on a number of the black squares. The players may take turns moving pieces, and there may be any number of additional players spectating the game.

Domain Area Detail



Application Architecture



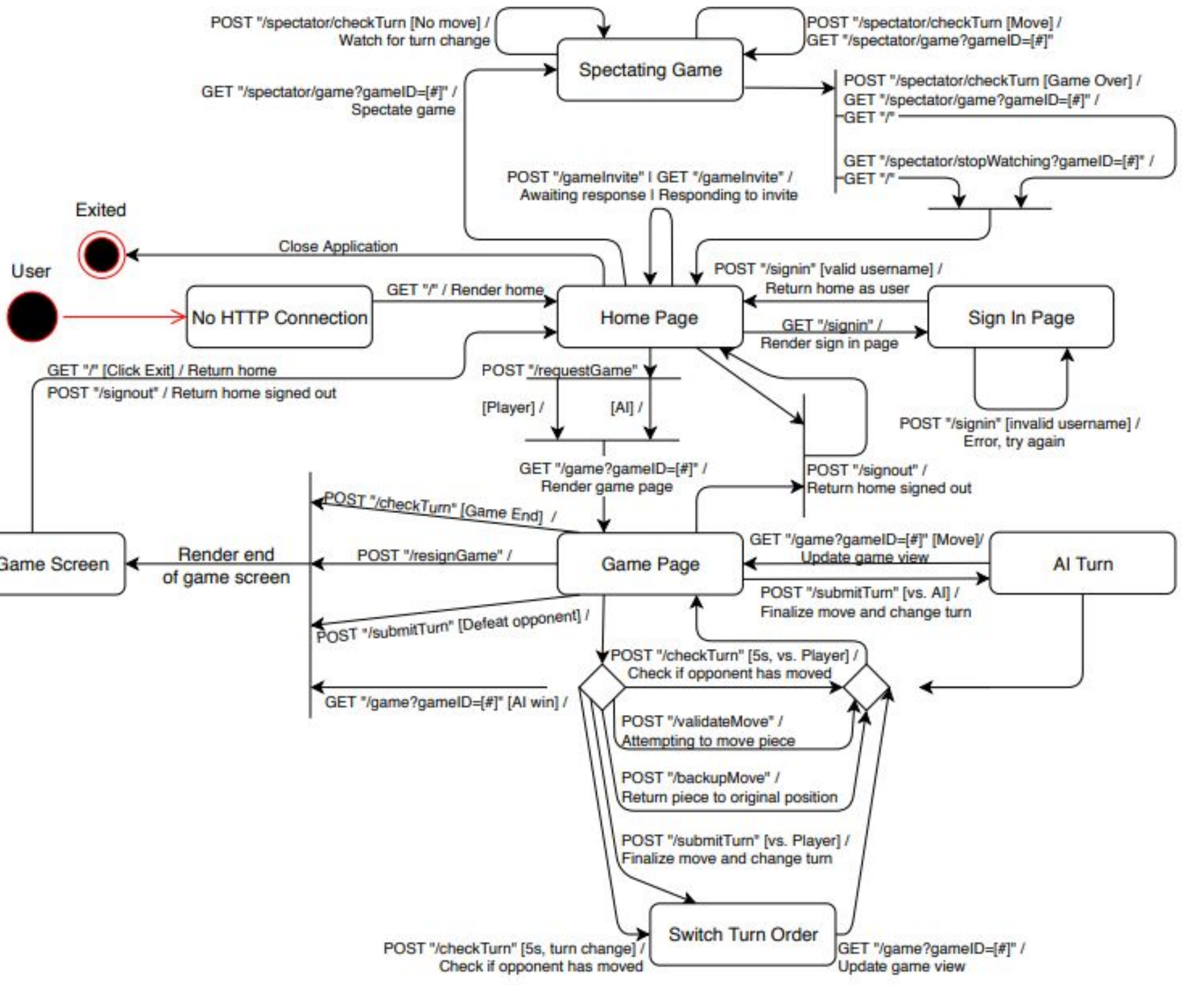
The player interacts with the game UI through their browser. On the client side is HTML, CSS for styling and graphics, and some Javascript to coordinate game actions and handle various form submissions.

On the server side, the application uses Spark framework to build UI controllers and Freemarker templates to build views. The application and model tiers are built with plain Java objects.

Summary:

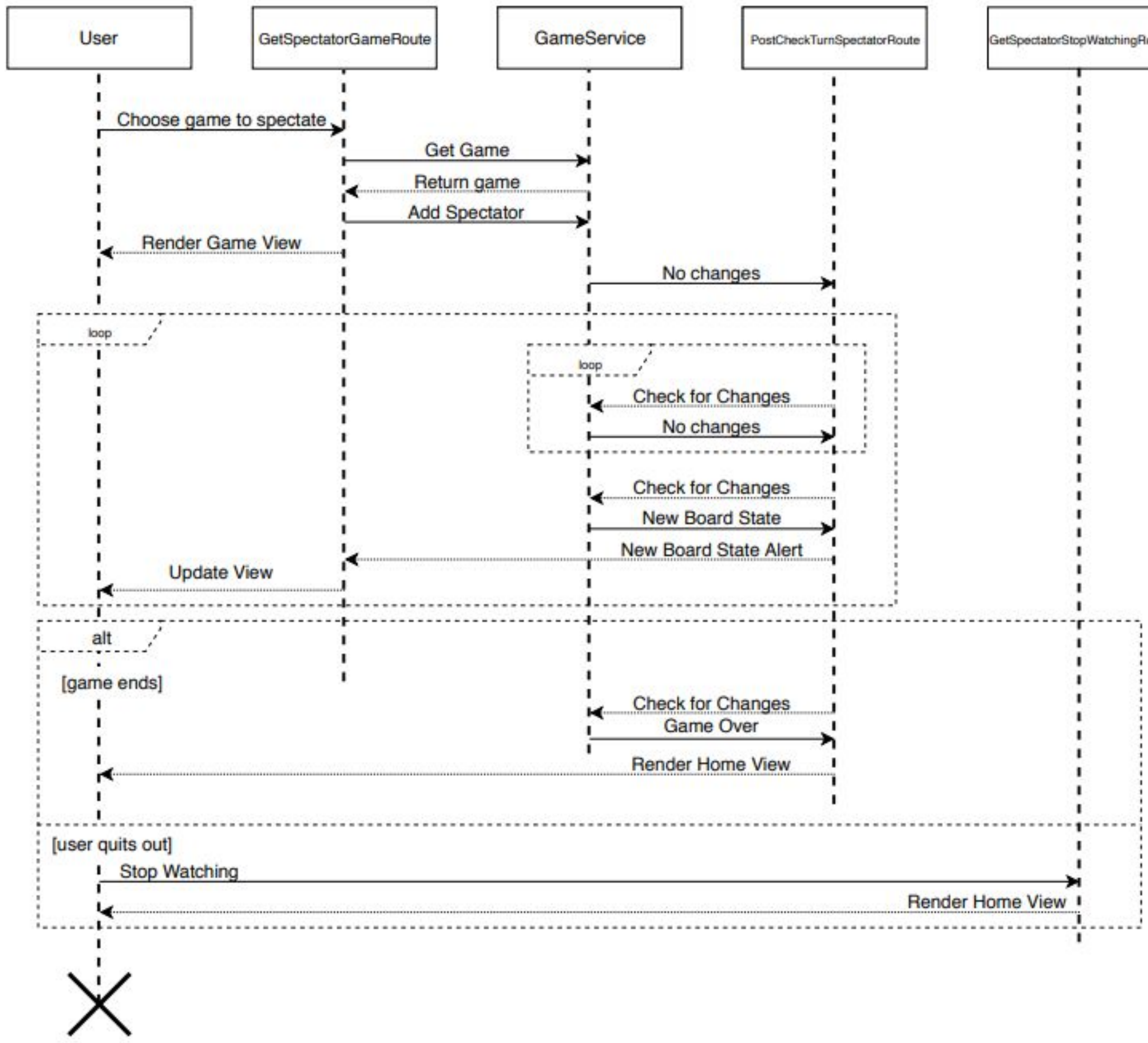
The program architecture is based on the above diagram. Firstly, the Server UI tier consists of a number of FreeMarker templates, which communicate via the controller to Java classes using the Spark framework. The Application tier runs these calls along with the Model tier, which models the Checkers Game board, pieces, and other gameplay requirements. The resulting program is displayed on a Web Browser, which is styled with CSS. The game itself is then run using a number of JavaScript classes.

Overview of User Interfaces:


POST "/spectator/checkTurn [No move] / Watch for turn change
Spectating Game
POST "/spectator/checkTurn [Move] / GET "/spectator/game?gameID=[#]"
GET "/spectator/game?gameID=[#]" / Spectate game
POST "/spectator/checkTurn [Game Over] / GET "/spectator/game?gameID=[#]" / GET "/"
GET "/spectator/stopWatching?gameID=[#]" / GET "/"
POST "/gameInvite" I GET "/gameInvite" / Awaiting response I Responding to invite
Exited
Close Application
User
No HTTP Connection
GET "/" / Render home
Home Page
POST "/signin" [valid username] / Return home as user
Sign In Page
GET "/signin" / Render sign in page
POST "/signin" [invalid username] / Error, try again
GET "/" [Click Exit] / Return home
POST "/signout" / Return home signed out
POST "/requestGame"
[Player] /
[AI] /
GET "/game?gameID=[#]" / Render game page
POST "/signout" / Return home signed out
POST "/checkTurn" [Game End] /
End Game Screen
Render end of game screen
POST "/resignGame" /
Game Page
GET "/game?gameID=[#]" [Move]/ Update game view
AI Turn
POST "/submitTurn" [vs. AI] / Finalize move and change turn
POST "/submitTurn" [Defeat opponent] /
POST "/checkTurn" [5s, vs. Player] / Check if opponent has moved
GET "/game?gameID=[#]" [AI win] /
POST "/validateMove" / Attempting to move piece
POST "/backupMove" / Return piece to original position
POST "/submitTurn" [vs. Player] / Finalize move and change turn
POST "/checkTurn" [5s, turn change] / Check if opponent has moved
Switch Turn Order
GET "/game?gameID=[#]" / Update game view

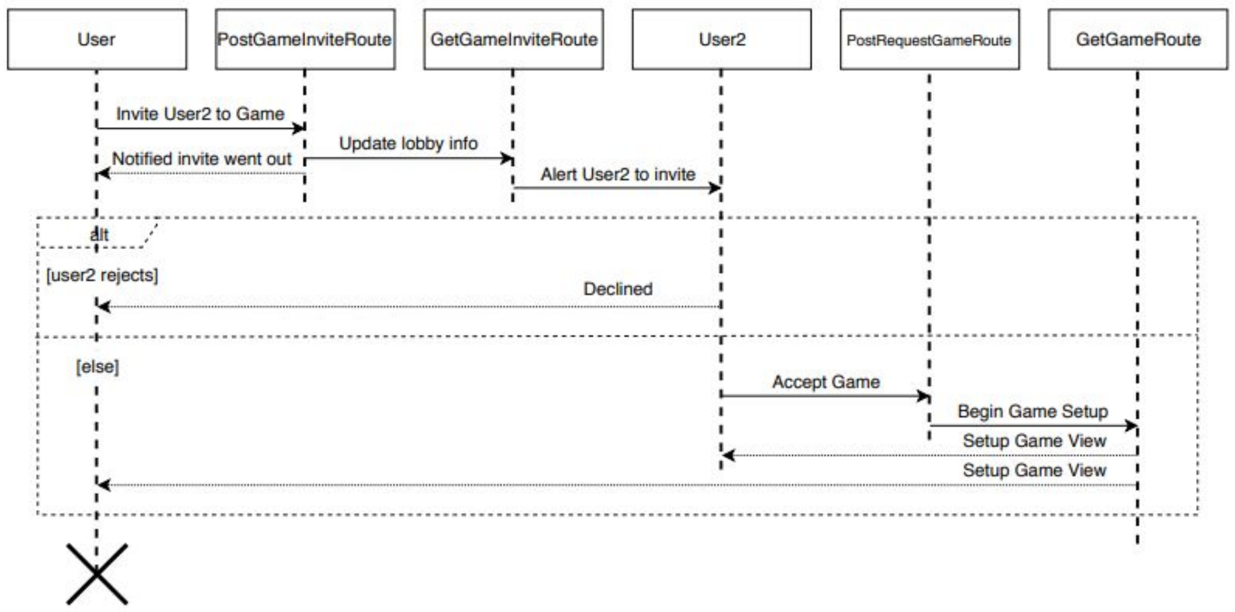
The UI tier is mainly comprised of classes implementing URL routes, including but not limited to: *GetGameRoute, GetHomeRoute, GetSignInRoute, PostValidateMoveRoute, etc…* These classes have a main functionality of causing a change in the player's view whenever a button is clicked or an action is taken in a game; however, *GetGameRoute* and *GetHomeRoute* have additional functionality of continuously updating the view of players viewing either the /home or /game routes. This functionality is necessary to bring a selected player into games on the home screen or switch turns during a game, as well as other actions that a player must wait for another player to make. The *PostCheckTurnRoute* also acts as a trigger for the spectator and the player whose turn is not currently active to receive the new board state from the /game route.

Subsystem: Spectating a Game



This diagram displays the process that the application goes through when a user is spectating a game. Initially, the user selects the game that they want to spectate, and the *GetSpectatorGameRoute* will retrieve the game and insert the user as a spectator to the game. As a spectator, periodically - every 5 seconds - the *GameService* method getBoardViewForSpectator is called to attempt to update the game view. While no new actions has occurred, nothing will happen; but, if a player makes a move, then *GetSpectatorGameRoute* will be called to update the view. Finally, there are two ways for the spectating session to end: the first is when the game ends naturally, the second is when the user selects the [Exit] button on the viewing screen. Both of the ways that the session ends will return the user back to the home route.

Subsystem: Starting a Game



This diagram displays a simplified view of starting a game with the reject a game invitation enhancement. Normally just selecting a valid player would start a game, but this enhancement allows the selected user to choose to decline the invitation. First, the initial user posts a request to the /gameInvite route which updates the lobby and alerts the second user. Then the user can either decline or accept the invite: declining just drops the game, while accepting posts to the /requestGame route and begins the formation of the WebCheckers game.

Overview of Application Interface:

The application interface consists of a PlayerLobby, GameLobby, GameService, and BoardService. The PlayerLobby service maintains a list of players signed into the application, and handles signing in, signing out, and retrieving players. The GameLobby service holds all ongoing games, and handles starting and ending games between players. The GameService works as an intermediary between other classes and the BoardService: it provides access to meta-information related to a game and actively switches the turn. The BoardService controls everything related to piece movement, including: basic moves, capture moves, consecutive capture moves, move validation, and backing-up and submitting moves.

Overview of Model Interface:

The model tier consists of a class to represent a user and various classes to represent the boardstate and view for the UI. The Player class represents a user in the game with methods to check the player user name and whether the player is currently in a game. The Board class is used to represent the internal state of the board: it offers methods for retrieving a piece at a particular space and method to move a piece. The Piece class represents a piece on the board and allows retrieval of the type and color.

The BoardView, Row, and Space classes are used in representing a view of the board to the UI. A new BoardView can be constructed to either be oriented towards red or white. A BoardView consists of 8 rows, each with 8 spaces that each allow for checking the piece on that space and whether the space is valid.

Testing:

This section will provide information about the testing performed and the results of the testing.

Acceptance Testing:

Acceptance testing for Sprint 1 included testing for user stories: *Player Sign-in* and *Start a Game*. Neither user story had issues in their fulfillment of the acceptance criteria and passed their tests on first review.

Acceptance testing for Sprint 2 included testing for user stories from Sprint 1 as well as user stories: *Move a Piece, Capture a Piece, King a Piece, Player Turns, Game Victory, Player Sign-out, Resigning from a Game* and *Reject Game Invitation*. We experienced issues in acceptance testing for stories *Player Sign-out* and *Resigning from a Game* due to incomplete knowledge of requirements when first defining the user stories - without referencing the *Additional Project Information* in the project resources, we made false assumptions about how a game should be ended that were discovered and rectified after testing.

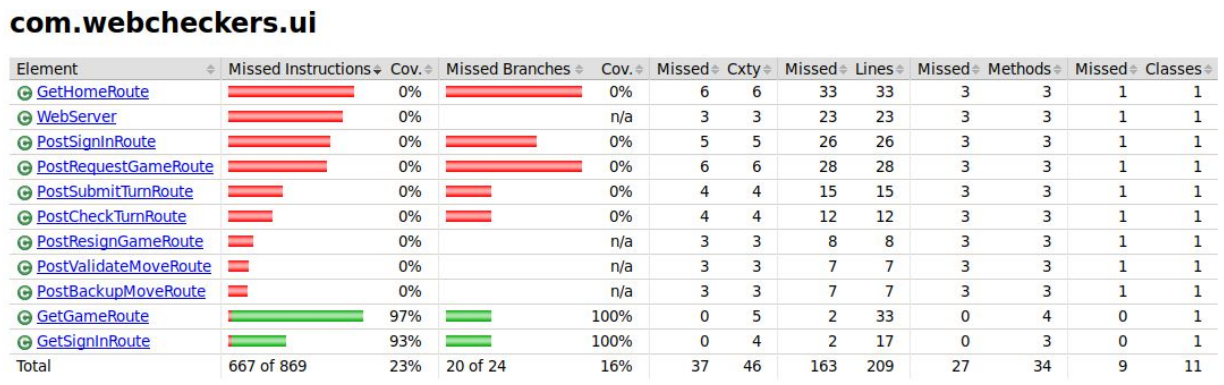
Acceptance testing for Sprint 3 included testing for user stories from Sprint 1 and Sprint 2 as well as user stories: *Basic AI, Advanced AI,* and *Spectate a Game*. Spectate game was relatively simple to implement after fixing an error with the *home.ftl* file which was restricted which players could play or spectate other players arbitrarily. Basic AI makes the first move it discovers to it was also simple to implement, but Advanced AI was making impossible moves for seemingly no reason until a new bug was discovered which allowed a player to make a simple move and a capture move in the same turn. After the bug was fixed, progress on Advanced AI continued relatively smoothly.

Unit Testing and Code Coverage:

Unit testing so far has only been completed for a small set of classes, as part of the class assignment. Moving forward, we plan to aim for 100% coverage by adding enough unit tests to cover all new code for any user story.

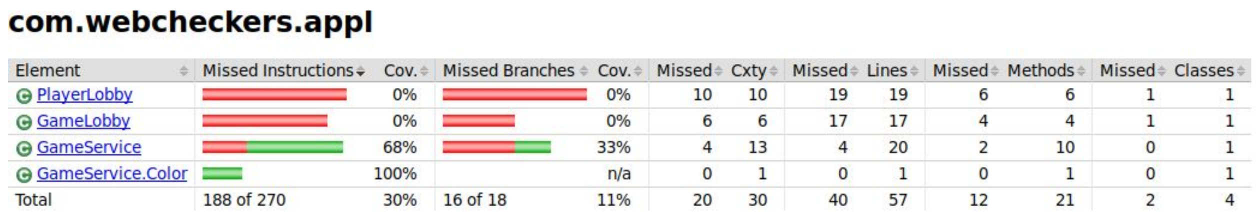
Code Coverage:

Code metrics were done using Maven, and slit up into UI, Appl, and Model tiers.



com.webcheckers.ui

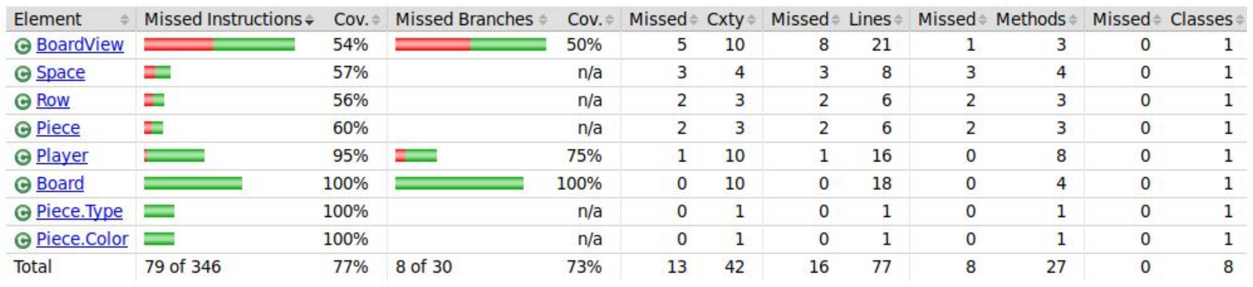
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
GetHomeRoute		0%		0%	6	6	33	33	3	3	1	1
WebServer		0%		n/a	3	3	23	23	3	3	1	1
PostSignInRoute		0%		0%	5	5	26	26	3	3	1	1
PostRequestGameRoute		0%		0%	6	6	28	28	3	3	1	1
PostSubmitTurnRoute		0%		0%	4	4	15	15	3	3	1	1
PostCheckTurnRoute		0%		0%	4	4	12	12	3	3	1	1
PostResignGameRoute		0%		n/a	3	3	8	8	3	3	1	1
PostValidateMoveRoute		0%		n/a	3	3	7	7	3	3	1	1
PostBackupMoveRoute		0%		n/a	3	3	7	7	3	3	1	1
GetGameRoute		97%		100%	0	5	2	33	0	4	0	1
GetSignInRoute		93%		100%	0	4	2	17	0	3	0	1
Total	667 of 869	23%	20 of 24	16%	37	46	163	209	27	34	9	11



com.webcheckers.appl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
PlayerLobby		0%		0%	10	10	19	19	6	6	1	1
GameLobby		0%		0%	6	6	17	17	4	4	1	1
GameService		68%		33%	4	13	4	20	2	10	0	1
GameService.Color		100%		n/a	0	1	0	1	0	1	0	1
Total	188 of 270	30%	16 of 18	11%	20	30	40	57	12	21	2	4

com.webcheckers.model

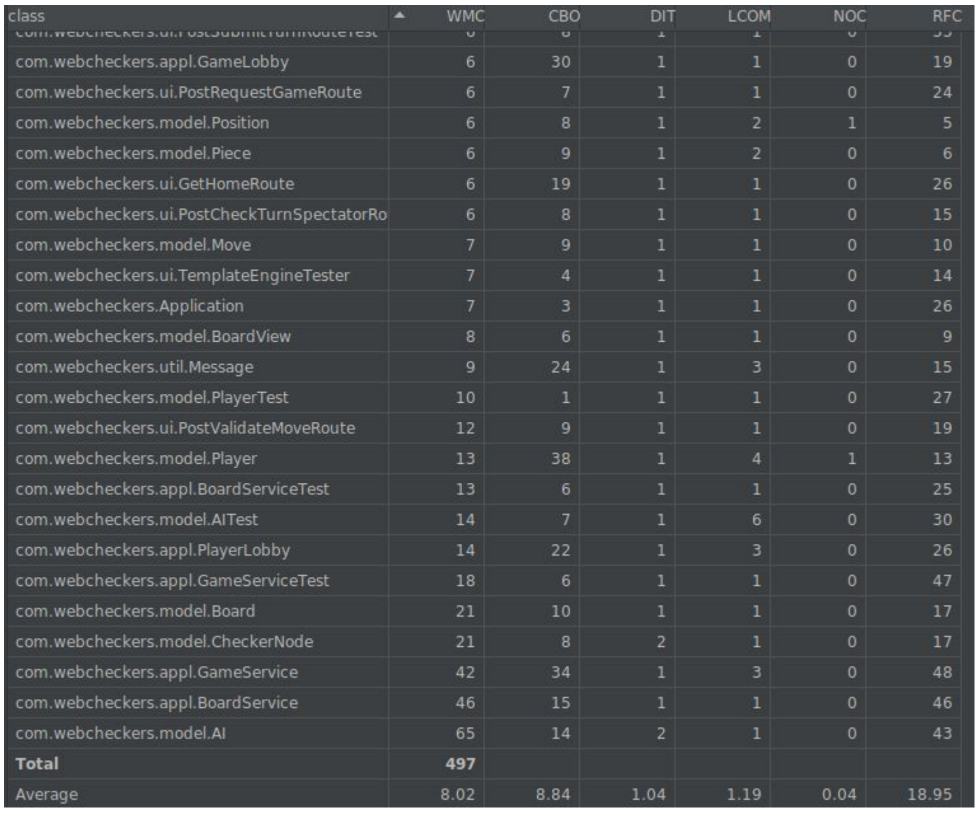


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BoardView	🟥🟩	54%	🟥🟩	50%	5	10	8	21	1	3	0	1
Space	🟥🟩	57%		n/a	3	4	3	8	3	4	0	1
Row	🟥🟩	56%		n/a	2	3	2	6	2	3	0	1
Piece	🟥🟩	60%		n/a	2	3	2	6	2	3	0	1
Player	🟩	95%	🟥🟩	75%	1	10	1	16	0	8	0	1
Board	🟩	100%	🟩	100%	0	10	0	18	0	4	0	1
Piece.Type	🟩	100%		n/a	0	1	0	1	0	1	0	1
Piece.Color	🟩	100%		n/a	0	1	0	1	0	1	0	1
Total	79 of 346	77%	8 of 30	73%	13	42	16	77	8	27	0	8

Code Metrics:

To analyze our code, we ran several metrics using the Intellij plugin and identified areas that did not meet the target on the class level.

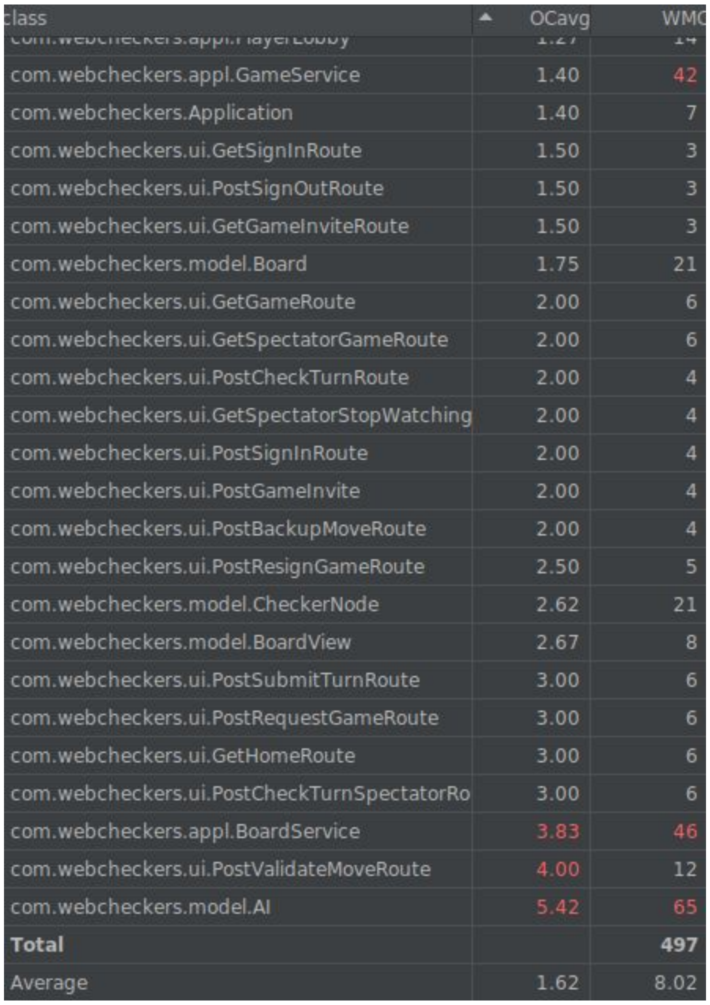
For Chidamber-Kimerer, we found no values that did not meet the target. Several classes had high coupling, but this was expected since they were critical objects within the system and thus used in many places. These classes include, for example: Player, GameService, and GameLobby. Below is a partial view of the results:



class	WMC	CBO	DIT	LCOM	NOC	RFC
com.webcheckers.appl.GameLobby	6	30	1	1	0	19
com.webcheckers.ui.PostRequestGameRoute	6	7	1	1	0	24
com.webcheckers.model.Position	6	8	1	2	1	5
com.webcheckers.model.Piece	6	9	1	2	0	6
com.webcheckers.ui.GetHomeRoute	6	19	1	1	0	26
com.webcheckers.ui.PostCheckTurnSpectatorRo	6	8	1	1	0	15
com.webcheckers.model.Move	7	9	1	1	0	10
com.webcheckers.ui.TemplateEngineTester	7	4	1	1	0	14
com.webcheckers.Application	7	3	1	1	0	26
com.webcheckers.model.BoardView	8	6	1	1	0	9
com.webcheckers.util.Message	9	24	1	3	0	15
com.webcheckers.model.PlayerTest	10	1	1	1	0	27
com.webcheckers.ui.PostValidateMoveRoute	12	9	1	1	0	19
com.webcheckers.model.Player	13	38	1	4	1	13
com.webcheckers.appl.BoardServiceTest	13	6	1	1	0	25
com.webcheckers.model.AITest	14	7	1	6	0	30
com.webcheckers.appl.PlayerLobby	14	22	1	3	0	26
com.webcheckers.appl.GameServiceTest	18	6	1	1	0	47
com.webcheckers.model.Board	21	10	1	1	0	17
com.webcheckers.model.CheckerNode	21	8	2	1	0	17
com.webcheckers.appl.GameService	42	34	1	3	0	48
com.webcheckers.appl.BoardService	46	15	1	1	0	46
com.webcheckers.model.AI	65	14	2	1	0	43
Total	**497**					
Average	8.02	8.84	1.04	1.19	0.04	18.95

Complexity Metrics:

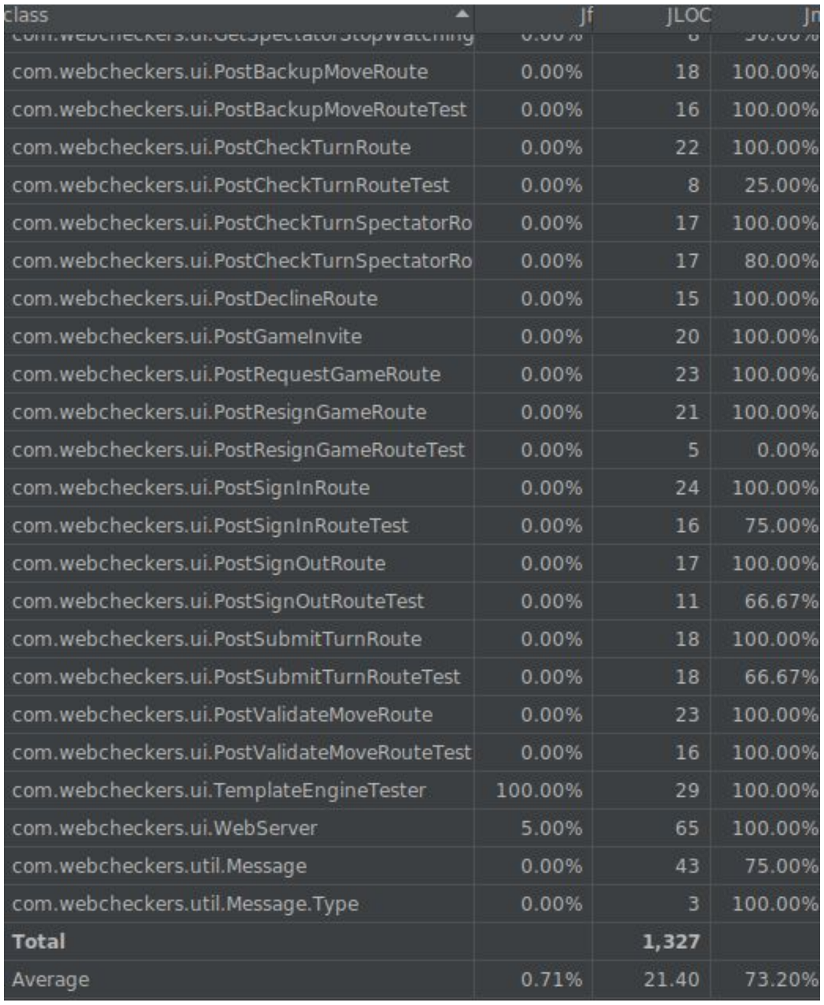
For complexity metrics, however, we discovered several classes that did not meet the target for at least one metric.



class	OCavg	WMC
com.webcheckers.appl.PlayerLobby	1.27	14
com.webcheckers.appl.GameService	1.40	42
com.webcheckers.Application	1.40	7
com.webcheckers.ui.GetSignInRoute	1.50	3
com.webcheckers.ui.PostSignOutRoute	1.50	3
com.webcheckers.ui.GetGameInviteRoute	1.50	3
com.webcheckers.model.Board	1.75	21
com.webcheckers.ui.GetGameRoute	2.00	6
com.webcheckers.ui.GetSpectatorGameRoute	2.00	6
com.webcheckers.ui.PostCheckTurnRoute	2.00	4
com.webcheckers.ui.GetSpectatorStopWatching	2.00	4
com.webcheckers.ui.PostSignInRoute	2.00	4
com.webcheckers.ui.PostGameInvite	2.00	4
com.webcheckers.ui.PostBackupMoveRoute	2.00	4
com.webcheckers.ui.PostResignGameRoute	2.50	5
com.webcheckers.model.CheckerNode	2.62	21
com.webcheckers.model.BoardView	2.67	8
com.webcheckers.ui.PostSubmitTurnRoute	3.00	6
com.webcheckers.ui.PostRequestGameRoute	3.00	6
com.webcheckers.ui.GetHomeRoute	3.00	6
com.webcheckers.ui.PostCheckTurnSpectatorRo	3.00	6
com.webcheckers.appl.BoardService	3.83	46
com.webcheckers.ui.PostValidateMoveRoute	4.00	12
com.webcheckers.model.AI	5.42	65
Total		**497**
Average	1.62	8.02

As we see above, there average operation complexity of BoardService, PostValidateMoveRoute, and AI are higher than the target. BoardService, AI, and GameService also have a high weighted method complexity.
Javadoc coverage metrics produced no outliers, which is expected since we consistently documented all our new classes and methods. Lines of code metrics also passed with no packages failing to meet the target value. Martin packaging metrics, too, produced no outliers. Below are the results from these 3 metrics.

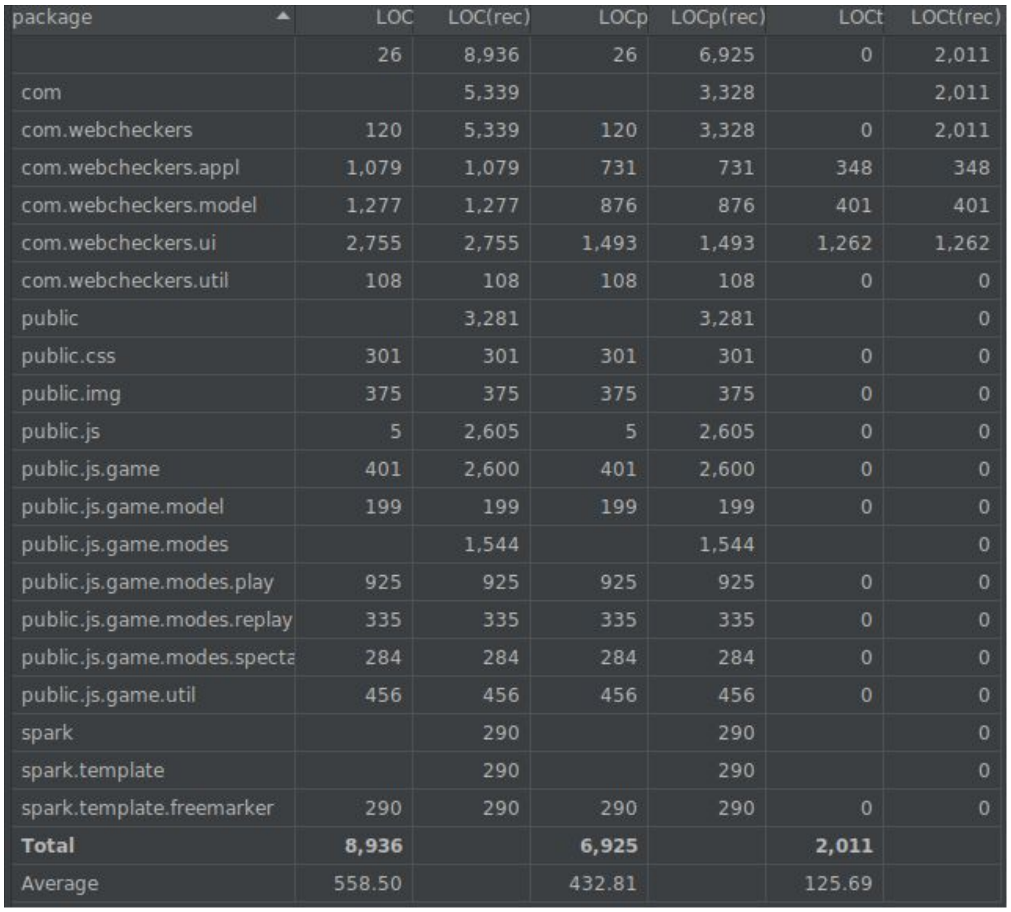
Javadocs:



class	Jf	JLOC	J
com.webcheckers.ui.GetSpectatorStopWatching	0.00%	8	50.00%
com.webcheckers.ui.PostBackupMoveRoute	0.00%	18	100.00%
com.webcheckers.ui.PostBackupMoveRouteTest	0.00%	16	100.00%
com.webcheckers.ui.PostCheckTurnRoute	0.00%	22	100.00%
com.webcheckers.ui.PostCheckTurnRouteTest	0.00%	8	25.00%
com.webcheckers.ui.PostCheckTurnSpectatorRo	0.00%	17	100.00%
com.webcheckers.ui.PostCheckTurnSpectatorRo	0.00%	17	80.00%
com.webcheckers.ui.PostDeclineRoute	0.00%	15	100.00%
com.webcheckers.ui.PostGameInvite	0.00%	20	100.00%
com.webcheckers.ui.PostRequestGameRoute	0.00%	23	100.00%
com.webcheckers.ui.PostResignGameRoute	0.00%	21	100.00%
com.webcheckers.ui.PostResignGameRouteTest	0.00%	5	0.00%
com.webcheckers.ui.PostSignInRoute	0.00%	24	100.00%
com.webcheckers.ui.PostSignInRouteTest	0.00%	16	75.00%
com.webcheckers.ui.PostSignOutRoute	0.00%	17	100.00%
com.webcheckers.ui.PostSignOutRouteTest	0.00%	11	66.67%
com.webcheckers.ui.PostSubmitTurnRoute	0.00%	18	100.00%
com.webcheckers.ui.PostSubmitTurnRouteTest	0.00%	18	66.67%
com.webcheckers.ui.PostValidateMoveRoute	0.00%	23	100.00%
com.webcheckers.ui.PostValidateMoveRouteTest	0.00%	16	100.00%
com.webcheckers.ui.TemplateEngineTester	100.00%	29	100.00%
com.webcheckers.ui.WebServer	5.00%	65	100.00%
com.webcheckers.util.Message	0.00%	43	75.00%
com.webcheckers.util.Message.Type	0.00%	3	100.00%
Total		**1,327**	
Average	0.71%	21.40	73.20%

Our method coverage is generally quite good, with a few missing. Our field coverage seems to come in at 0%. We believe this is because, unfortunately, the plugin only counts those parameters which have a “{link}” tag included. This metric can therefore not be relied upon. Outside of that, the coverage is decent, with a few missing comments in some classes causing it to go down.

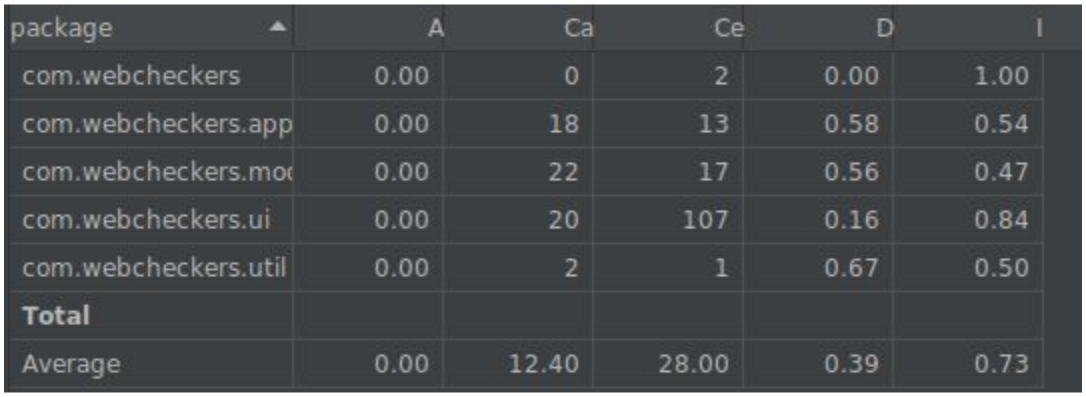
Lines of Code:



package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
	26	8,936	26	6,925	0	2,011
com		5,339		3,328		2,011
com.webcheckers	120	5,339	120	3,328	0	2,011
com.webcheckers.appl	1,079	1,079	731	731	348	348
com.webcheckers.model	1,277	1,277	876	876	401	401
com.webcheckers.ui	2,755	2,755	1,493	1,493	1,262	1,262
com.webcheckers.util	108	108	108	108	0	0
public		3,281		3,281		0
public.css	301	301	301	301	0	0
public.img	375	375	375	375	0	0
public.js	5	2,605	5	2,605	0	0
public.js.game	401	2,600	401	2,600	0	0
public.js.game.model	199	199	199	199	0	0
public.js.game.modes		1,544		1,544		0
public.js.game.modes.play	925	925	925	925	0	0
public.js.game.modes.replay	335	335	335	335	0	0
public.js.game.modes.specta	284	284	284	284	0	0
public.js.game.util	456	456	456	456	0	0
spark		290		290		0
spark.template		290		290		0
spark.template.freemarker	290	290	290	290	0	0
Total	**8,936**		**6,925**		**2,011**	
Average	558.50		432.81		125.69	

This metric shows the distribution of lines of code throughout the tiers of the application. Interestingly, the UI tier contains the most lines, and the Application tier contains the least. This is probably due to the UI and Model tiers having a larger number of classes than the Application tier.
We can also see from these metrics that the number of lines of test code is somewhat commensurate with the number of lines of code.

Martin Packaging:



package	A	Ca	Ce	D	I
com.webcheckers	0.00	0	2	0.00	1.00
com.webcheckers.app	0.00	18	13	0.58	0.54
com.webcheckers.mo	0.00	22	17	0.56	0.47
com.webcheckers.ui	0.00	20	107	0.16	0.84
com.webcheckers.util	0.00	2	1	0.67	0.50
Total					
Average	0.00	12.40	28.00	0.39	0.73

The above shows the metrics on the package level. Abstractness is at 0 since we did not use interfaces or subclassing in this project. Afferent coupling is relatively consistent between classes. Efferent coupling, however, is very high in the UI tier, which might be expected since the UI has many dependencies on application components it needs in order to coordinate and fulfill requests.
As a result, the instability of UI is also fairly high compared to the instability of the other two tiers.

Design Quality and Possible Revisions:

Our team was fairly happy with the overall quality of the design for this product. Throughout the development process, it was easy to add new features (like spectate and AI) without having to overhaul any significant portion of the code, and that is an indication of the quality of our design. We believe that the strong points of our design lie in the separation of responsibilities between classes (using the single responsibility principle), the simplicity of logic in the UI tier and the controllers, and keeping the complexity of individual methods low in most classes.

There are definitely areas for improvement, however. From the code metrics, it is clear that complexity in some classes is high. The two classes most in need of a reduction in complexity are BoardService and AI.

BoardService is complex due to all the logic needed for the game. It handles things like move validation, turn validation, and moving pieces. All of this was implemented using simple control flows and without any helper classes. To reduce complexity in this class, we would want to introduce several classes to handle internal logic, such as a MoveValidator or TurnValidator. We could use Liskov substitution in order to have 2 different validators of each type, one for red and one for white, call one depending on whose turn it is. Each validator would then have a smaller set of possible moves to worry about, and there would be fewer nested if-else control flow statements. This would make BoardService significantly less complex and probably make it meet the target for the complexity metrics.

AI is the other class most in need of a reduction in complexity. One obvious improvement would be to try to separate out the code into multiple classes. We used the minimax algorithm, for example, and this algorithm could be put into a separate class. The other methods in AI could also be put into separate classes that will handle all the internal logic, such as for calculating the value of a board state for the minimax algorithm, or for moving pieces on the board to produce new states.

Another improvement we would want to make is trying to make certain classes more testable. This was a particular problem with BoardService and AI, because it was difficult to test the methods in them since they relied so much on certain board states. To improve this, we might want to use principles like dependency injection in order to make it easier to create a mock class with predictable behavior that we pass into those classes when testing. In this way, it’s much easier to unit test them and verify correct functionality.