Systems Design Document for Zcrabble by Scrabblers

Niklas Axelsson, Martin Björklund, Ole Fjeldså, Gustaf Jonasson
Date: 2021-10-24
Version: Final

1 Introduction

This document aims to give an overview of how the Zcrabble game works. It contains an explanation of Zcrabble specific terms, system architecture and system design. Further, the document also contains information about dependencies and how data management is handled.

Zcrabble is a game very reminiscent of the popular game Scrabble. It is designed to be played by two to four players. Players take turns creating words based on a few letter tiles that the players have at their disposal. Points are awarded, based on a set of rules, after each turn. When the game ends, the player with the most points is victorious.

1.1 Definitions, acronyms, and abbreviations

This section outlines a number of Scrabble specific terms.

Play: In the context of the game, a play is the action of placing a word on the board and confirming it by ending one's turn.

Cell/Square: Cells or squares are used interchangeably and refer to the spaces on the board segmented into x*x sizes depending on the board. Specifically a square is a single one unit of space where a letter tile is placed. For example a 2 by 2 sized board would fit 4 squares.

Special cell/square: Some cells/squares are worth additional points or have other modifiers when a tile is placed on them.

Board: A board is the total collection of all squares.

Letter tile: A letter tile or shortened to “tile”, fits on a cell/square and has a single character belonging to whichever alphabet the dictionary is made up of. A blank tile is also a valid tile. A tile also has a score associated with each letter.

Rack: A collection of letter tiles that belong to the player and are the tiles the player can choose to play during their turn. A player may also choose to exchange any number of tiles from the rack for an equal amount of new ones from the letter tile bag.

Tile bag: A container used by players to fill the rack with new letter tiles at the start of their turns. The game is over when the bag is empty and one player's rack is empty.

Word: Specifically in the context of the game, a game "word" is any series of letters formed by letter tiles that can be found in whichever dictionary or sets of words the game uses to reference. A word can be read vertically or horizontally but not diagonally. A word is always read from left to right or from up to down.

Score: Each player has a score associated with them which is their total accumulation of points by creating words over the course of the game. The player with the highest score wins.

2 System architecture

At startup, JavaFX initialization is called and the start screen is opened. After the start button is pressed the main window is opened and a game is launched. The user can then decide to play the default game (one human player versus one bot) or create a new game from the menu at the top. Creating a new game communicates from the controller to the model to start a new game.

When a new game is started, the tiles to use and what board to use is read from text files. The "game loop" starts here and each player takes their turn placing tiles on the board and then pressing end turn to pass the turn to the next player. At the end of each turn, the current player draws up to 7 tiles on their rack from the tile bag. When tiles are placed on the board they are put in a temporary board in the model. Once the player ends their turn the model either accepts the new word and places it on the permanent board, gives out a score and passes the turn or denies it and the current player can try again. The player can also choose to exchange up to 7 of their tiles with new tiles from the tile bag, this ends that player's turn. All these user interactions (pressing buttons, moving tiles) are noticed by the controllers and communicated to the model.

Once the tile bag runs out of tiles and a player has used up their rack or all the players pass their turn without a word for 7 consecutive turns the game ends. The user can then choose to start a new game from the menu or quit the game.

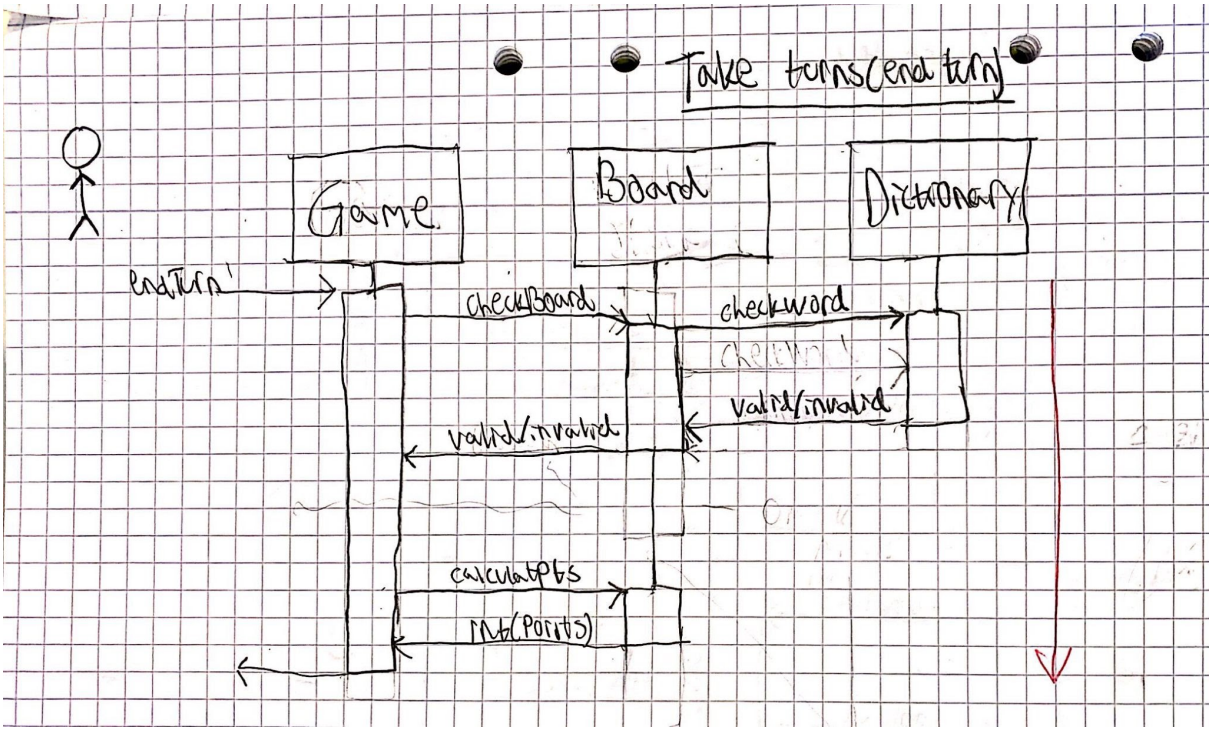

Figure 1: Sequence diagram visualizing the “Take turns” user story. This version is an early draft from september 2021.

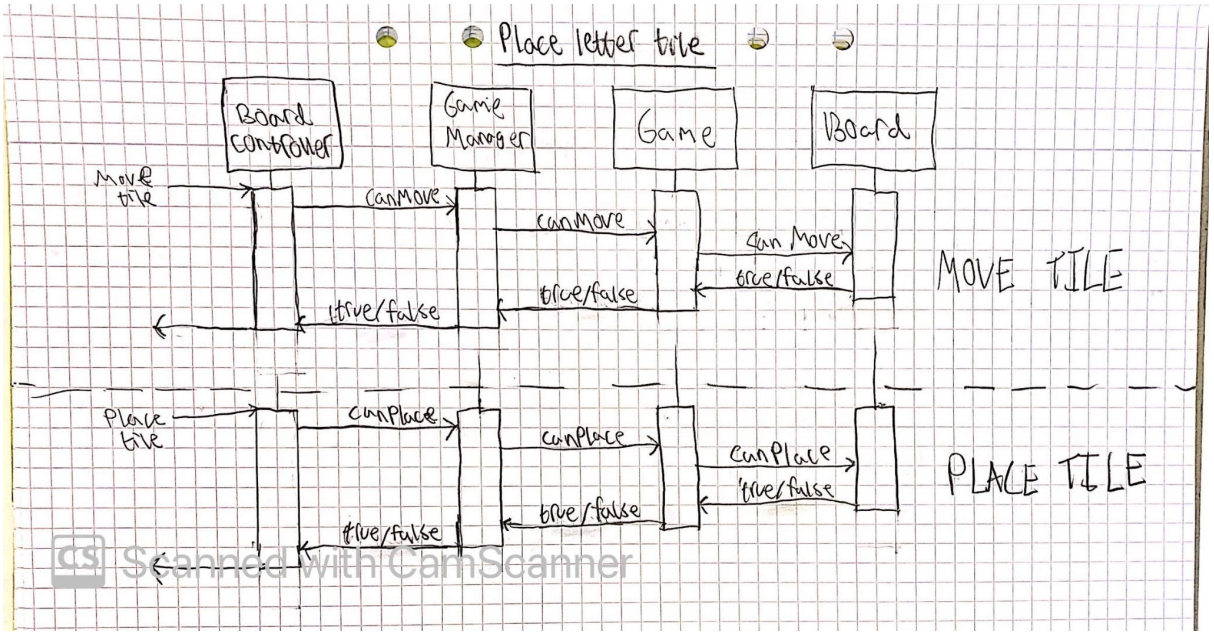

Figure 2: Sequence diagram visualizing the “Place letter tile” user story. This version is an early draft from september 2021 and is no longer representative of how the game implements this functionality.

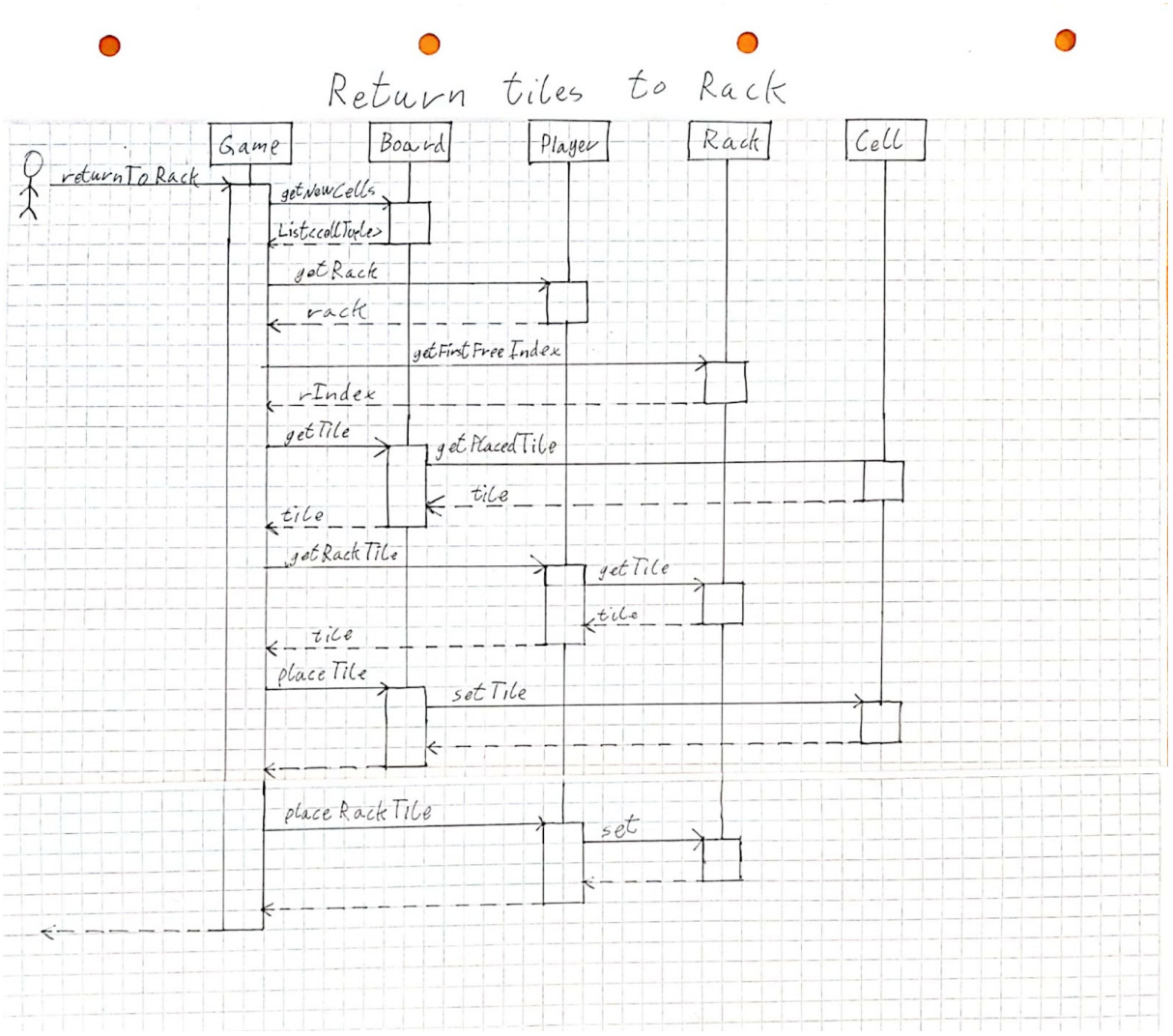

Figure 3: Sequence diagram visualizing the “Return tiles to Rack” user story. This is the implementation from 2021-10-22.

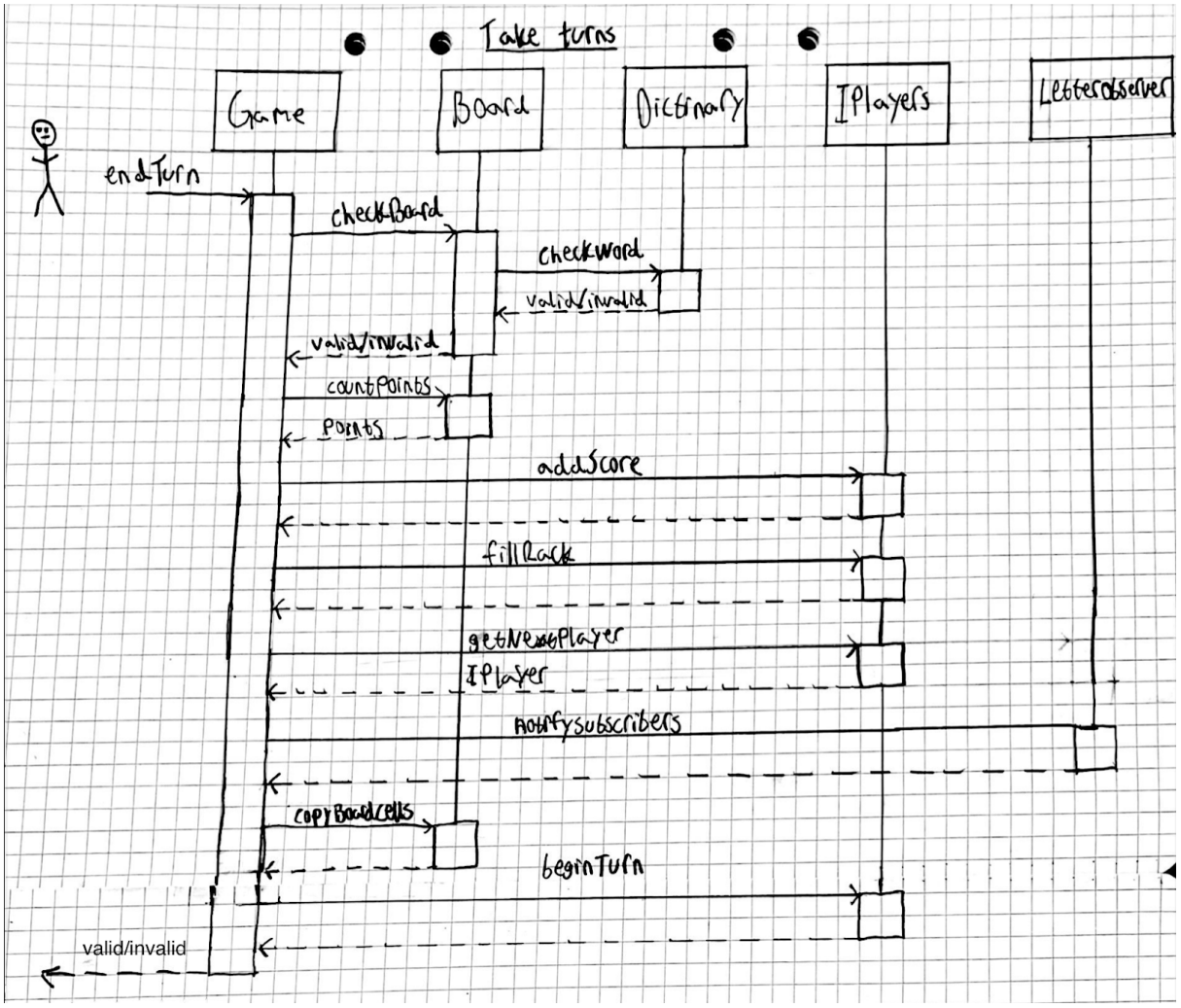

Figure 4: Sequence diagram visualizing the “Take turns” user story. This is the final version.

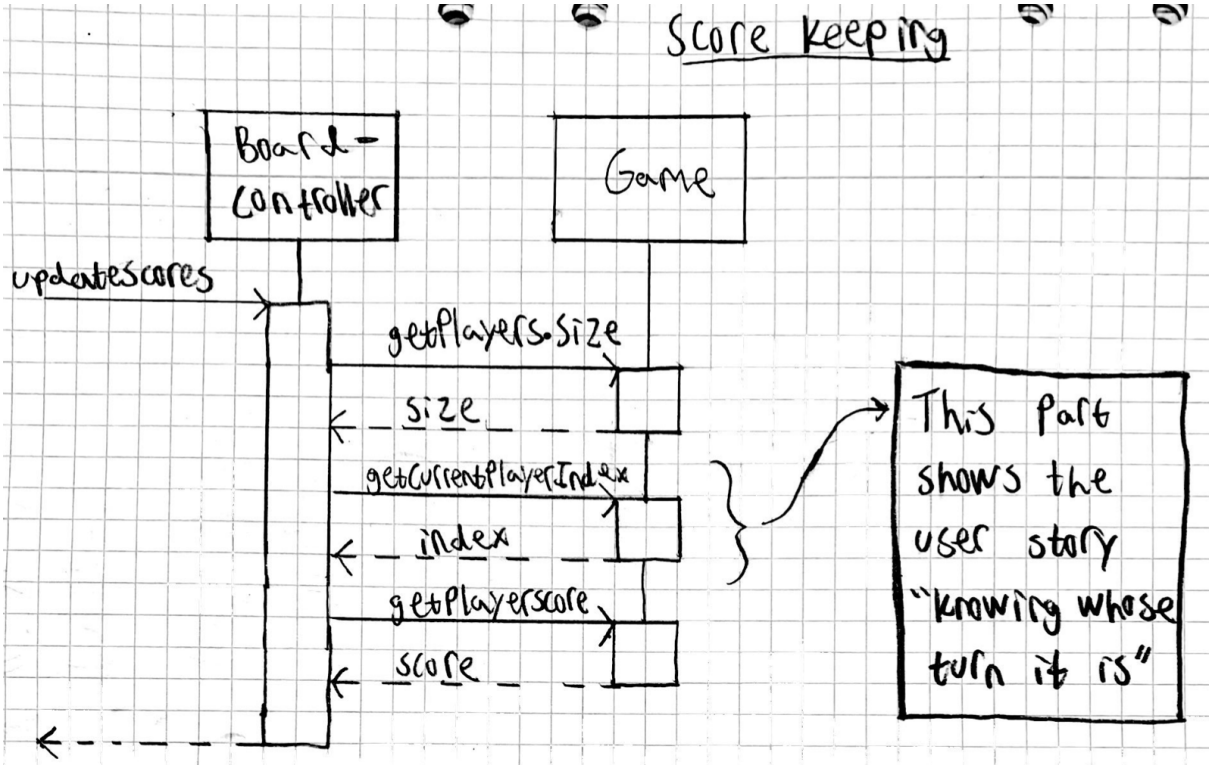

Figure 5: Sequence diagram showing the “Score keeping” user story. The middle part also visualizes the “Knowing whose turn it is” user story.

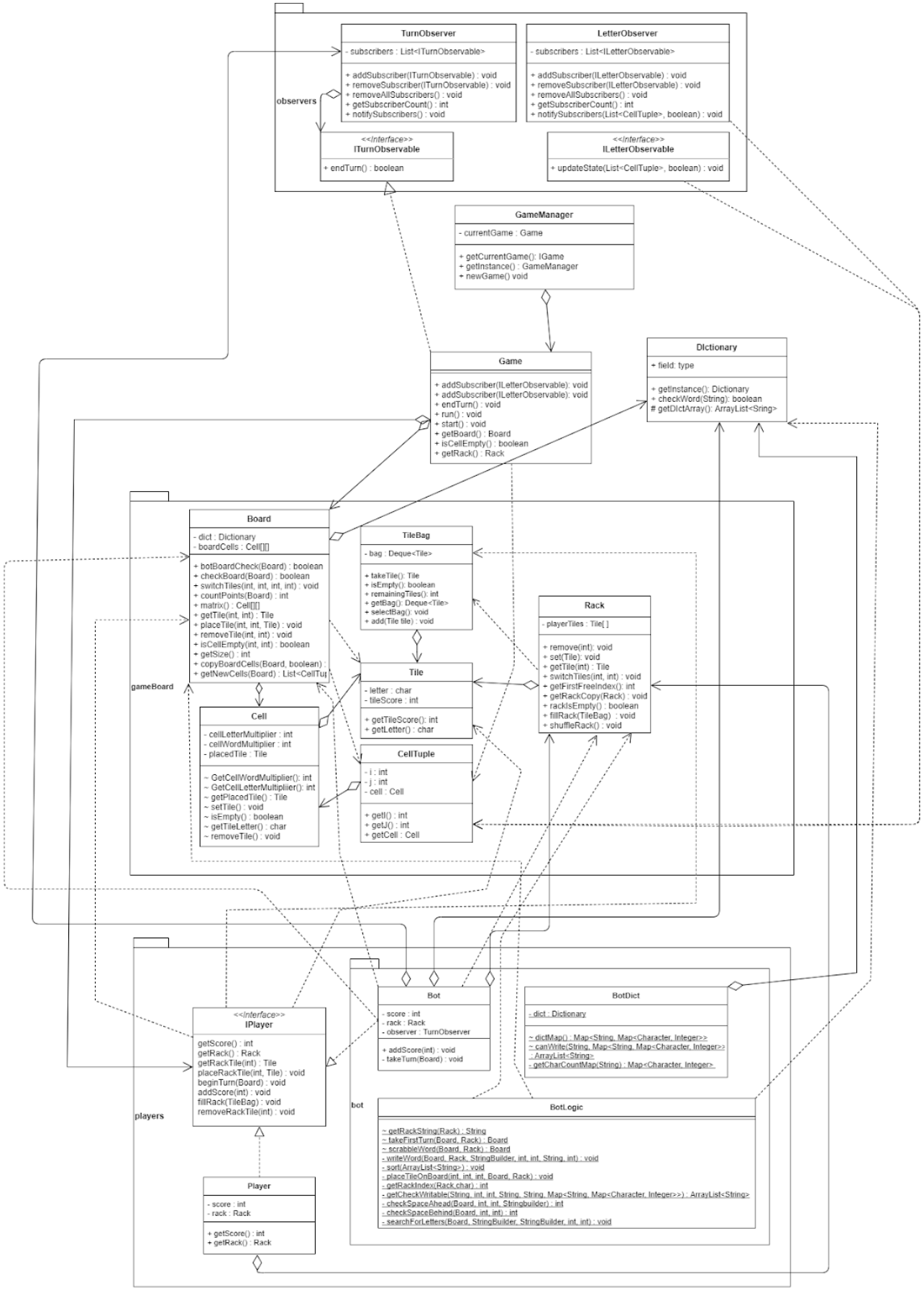

Figure 6: Design model.

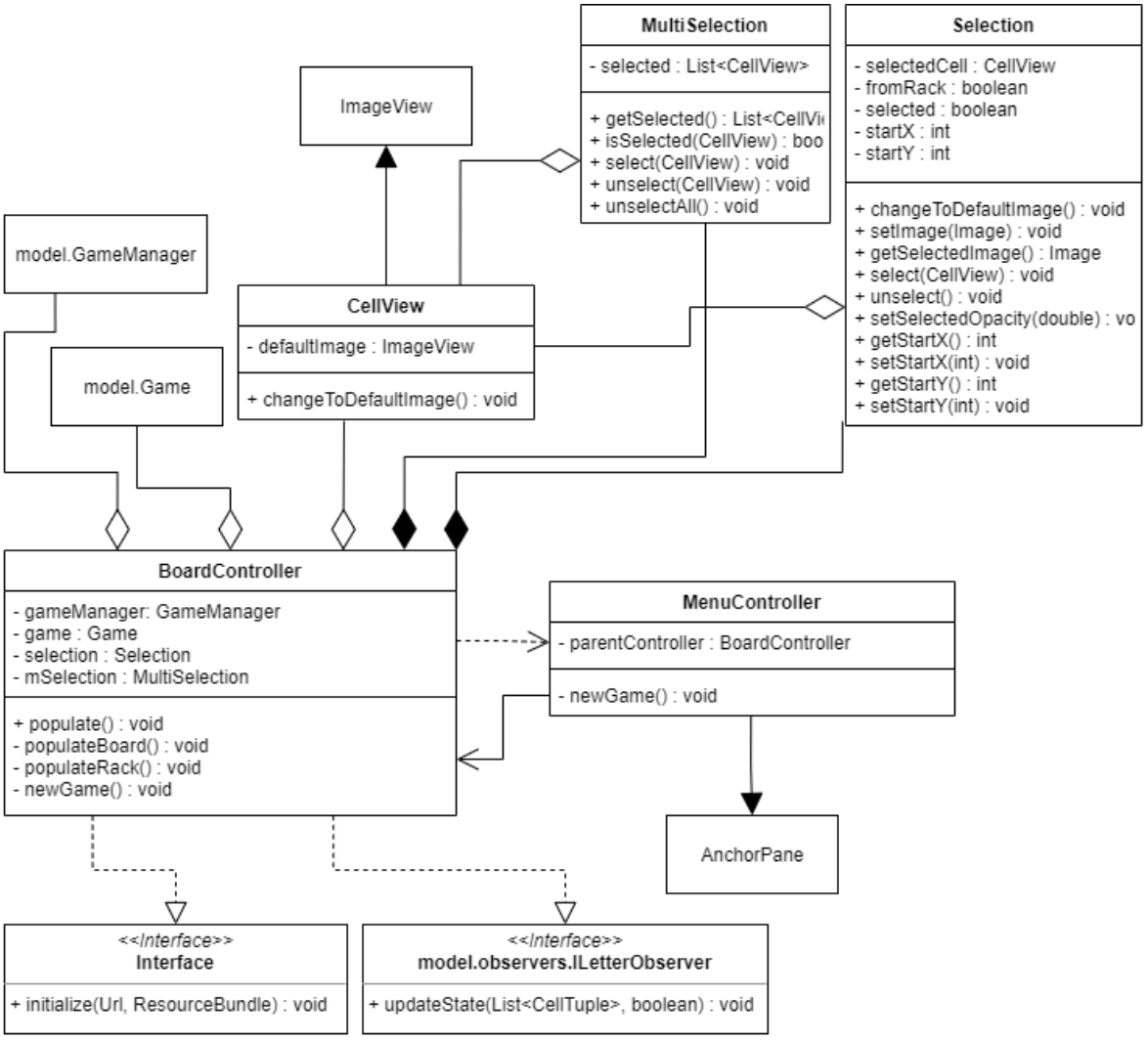

Figure 7: UML-diagram for the controller package.

3 System design

MVC

JavaFX and SceneBuilder were used to handle the graphics of the application and thus its protocols had to be followed. The application's GUI was designed in Scenebuilder and JavaFX imports a .fxml file to load the scene. Controller classes were connected to the .fxml and to GUI elements in the scene and the class is initialised when that .fxml is loaded.

It could be argued that the fxml files function as different views and since it is only possible to access the GUI elements through those classes the view and controller parts of MVC are very tightly coupled. The model however is separated as much as possible from the other modules.

The controllers that need to communicate with the model have a reference to a Game and calls the method it needs and influences the model directly. In order to avoid a circular dependency between the model and the view/controller an observer pattern was used. If the model has a change that impacts the view it publishes the changes to the views that are listening.

Relation to domain model

The design model has a very close match to the domain model. Everything in the domain is represented by a class in the design. So there is a very clear relation between each item in the domain model and its corresponding one in the design model. Since the things we included in the domain model are actual things/nouns it went well to incorporate them into the design. Some things turned out to be too big to fit in a single class so they became their own sub packages in the model, e.g. bot.

Design Patterns

The observer pattern was used twice in the code, used to update the view about changes in the model. The Game class works as a facade pattern and an entry point into the model. The singleton pattern is used to easily gain access to Game, EmptyTile and Dictionary, both of which only one instance of is needed. The codebase also uses a singleton for an enum called RandomSeed. This enum is used with a fixed seed for testing.

4 Persistent data management

- Text documents
- Icons

The game reads the data for how to populate the bag and the board from txt files. The dictionary is read from a .txt file upon starting the program.

5 Quality

Known Issues/Possible improvements:

- Players should not keep track of their own Rack and Score, there are currently problems because Rack belongs to a player but needs to be in the gameBoard package to access Tile and Cell. This creates an inconsistency in how Game communicates with the Rack. See sequence diagram “Return tiles to rack” for details.
  If the player's Rack was handled by Game there would be no need for most Rack operations to go through Player.
- Bot and Board are tightly coupled.

- When the bot is calculating, it blocks the thread and the UI stops being interactable, so running the bot/model in its own thread would be a good idea.
- Make more classes in gameBoard (e.g. Tile) be less visible, i.e make methods package private.
- Implement a singleton for an empty tile. There is currently a class called EmptyTile, but it is not used throughout the codebase.

Testing:

- Testing is done using JUnit. The tests can be found in the src/tests folder. The code coverage of the model package is at 96%.

JDepend:

- Figure 8 shows the JDepend report.

Package	TC	CC	AC	Ca	Ce	A	I	D	V
com.zcrabblers.zcrabble	1	1	0	0	11	0.0%	100.0%	0.0%	1
com.zcrabblers.zcrabble.controller	3	3	0	0	5	0.0%	100.0%	0.0%	1
com.zcrabblers.zcrabble.model	3	3	0	2	6	0.0%	75.0%	25.0%	1
com.zcrabblers.zcrabble.model.gameBoard	7	7	0	3	5	0.0%	62.0%	38.0%	1
com.zcrabblers.zcrabble.model.observers	2	0	2	2	2	100.0%	50.0%	50.0%	1
com.zcrabblers.zcrabble.model.players	2	1	1	2	2	50.0%	50.0%	0.0%	1
com.zcrabblers.zcrabble.model.players.bot	4	4	0	1	6	0.0%	86.0%	14.0%	1
com.zcrabblers.zcrabble.utils	1	1	0	1	3	0.0%	75.0%	25.0%	1

Figure 8: JDepend report.

5.1 Access control and security

- Does not apply to this project.

6 References

- Zcrabble rules can be found in project resources.
    - Zcrabble has some rule changes from the original Scrabble.
        - There are no wildcards in Zcrabble, instead there is an extra “Z” and “Q”.
        - It is not possible to challenge other players' words in Zcrabble instead the game automatically checks every play and refuses invalid words.
        - If an incorrect play is made the player does not lose their turn but is instead allowed to make another play or skip their round.
- This codebase uses JavaFX to represent the application graphically.
- Testing was done using JUnit.
- Maven is the build automation tool of choice.
- This codebase supports JDepend.
    - Run “mvn site” then “open target/site/jdepend-report/html” to read the report. Alternatively, see section 5 for a picture of the report.