CS 1530 – SPRINT 3 DELIVERABLE

27 Oct 2016

David Bickford (drb56@pitt.edu)

Craig Kodman (cmk126@pitt.edu)

Joe Meszar (jwm54@pitt.edu)

David Tsui (dat83@pitt.edu)

SCRUM MASTER
David Bickford

Accomplishments

This sprint came together quite well for us, giving us the best velocity we have had since the start of our project. Our architectural model focuses around the Model-View-Controller (MVC) software design pattern where we have a Model (our programmatic chessboard implementation), a View (our GUI), and a controller (The code that handles the GUI actions by interacting with our Model). In this sprint, we were finally able to integrate the View with the back-end model through the near-completion of our document controller. This included creating an internal 2D array that correlated with the chessboard within the GUI. Each component of the array is a **char** value that holds the value of which chess piece it contains (‘`PNRBQK`’ for White pieces, ‘`pnrbqk`’ for Black pieces, and *`null`* for empty squares). With this 2D array representing the chessboard, we were able to incorporate it into a `ChessBoard` class that has helper methods for manipulating the chessboard. We can now move chess pieces legally according to the USCF standards using Standard Algebraic Notation (SAN) based off the user moving a piece from one chess square to another; we have created valid FEN strings representing the current state of the chessboard; the logic is now in place to only take chess pieces when a White piece has been moved to a square with a Black piece (and vice versa); and finally, SAN strings are now being converted properly to 2D array notation (‘`a1`’ to ‘`70`’).

These core functions of the program were created through collaboration across all team member branches on Github. This collaboration was performed in-person (groups), and also via an electronic chat service called Slack. Communication in this format has been the de facto standard for our group since we began our project.

Disagreements came about when we went to implement the conversion from SAN strings to 2D array coordinates--half of our group thought we should implement the 2D array as `[column][row]` to correspond with the SAN format of `[column][row]`, but this goes against the programming nature of how a 2D array is called. In the end, we came to the conclusion that translation functions were a better option, and then to **only talk in terms of SAN** (‘`a1`’) instead of confusing ourselves with 2D coordinates (‘`70`’). Another problem arose with the syntax of our programming styles:

STYLE #1	**STYLE #2**
`if () {` `} else {` `}`	`if ()` `{` `}` `else` `{` `}`

We chose to go with `Style #1`, just based on the fact that most of our code was already written in this manner.

The main challenge with our code this sprint involved combining it all together and making it work. Each member had a specific task to perform, and that task was dependent on another member’s task. This circular reference created issues where tests could not be implemented right away and caused panic for everyone, making us all think that our code was not going to work properly. In the end, our methods worked as expected and only needed minor tweaking (for example which parameters to pass in, and what exactly needed to be returned) to completely integrate together.

User Stories Completed

(***#6***) As a User, I want to move my chess pieces, so that I can play chess.

We decided on this user story because we needed to be able to move the chess pieces and actually test the ability to generate the appropriate FEN strings when a move was made.
(**16 points**)

(***#7***) As a User, I want to abide by the United States Chess Federation (USCF) Rules, so that I can play a regulation game of chess.

We decided on this user story because we wanted to test the chess engine and have a working chess game by the end of this sprint.
(**16 points**)

(***#8***) As a User, I want to be warned of illegal moves, so that I can make only legal USCF moves.

We decided on this user story because we wanted to have a working chess engine by the end of this sprint.
(**16 points**)

Velocity: **48**

Github Link

— https://github.com/drb56/CS1530 —

Defects Found

1. **Problem**: FEN string was not printing out correctly.

   Solution: We had a situation where we were actually updating the board in two different places, so the FEN string was not displaying the correct output. It would display the movement of where the piece came from but not update the location the piece moved to.

   Reproduction steps: Make the `update2DArrayChessboard` method public and call it elsewhere within `LaboonChessDocumentController.java`.

   Expected behavior: We expect that the board will be updated correctly regardless of the number of moves the user plays.

   Observed behavior: We get a FEN string that is not accurate to the current board state.

---

2. **Problem**: Promoting a pawn

   Solution: We have yet to create a solution for this problem. We decided to push this off until next sprint because we ran out of time.

---

3. **Problem**: Not successfully showing when king is in check or checkmate

   Solution: We have yet to create a solution for this problem. We decided to put this off until next sprint because we ran out of time.

---

4. **Problem**: Refactored our code, and the `sanTo2DCol` method, which was passing our tests, was now failing them.

   Solution: Rewrote `sanTo2DCol` so that the it returned `-1` when the column number was less than `0` and greater than `7`, instead of less than `0` and greater than `8`.

   Reproduction steps: Run gradle test

Expected behavior: `testSanTo2DColOutOfBoundsUpper` and `testSanTo2DColOutOfBoundsLower` should pass by having `sanTo2DCol` return `-1`.

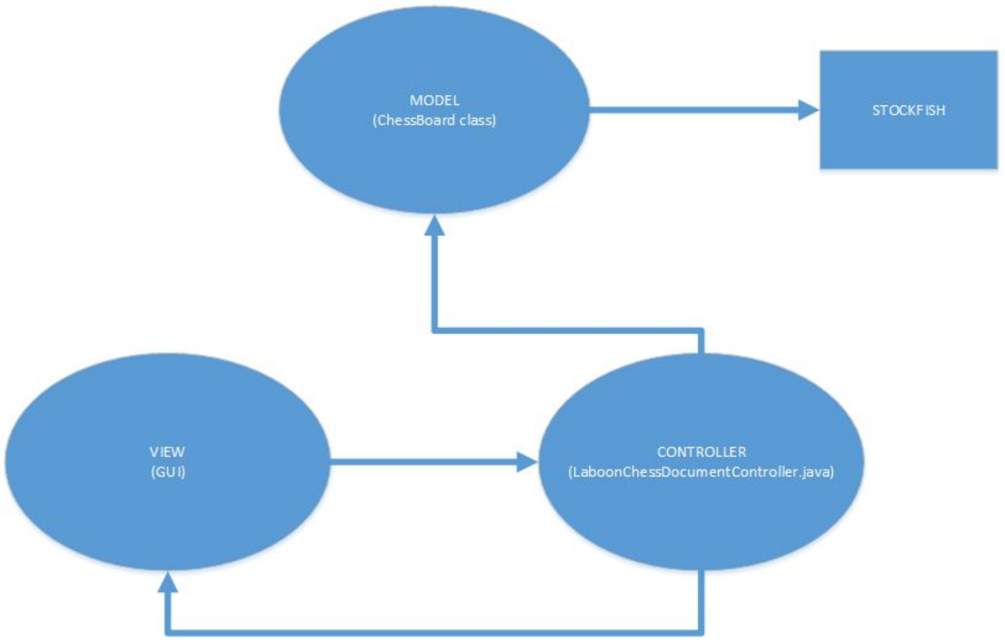
Observed behavior: `0` was returned by `sanTo2DCol` when `testSanTo2DColOutOfBoundsUpper` and `testSanTo2DColOutOfBoundsLower` were ran causing both tests to fail.

This sprint we primarily focused on giving the chess game the ability to legally move chess pieces and making sure the chess pieces can move in only the way they are supposed to. As a result, we only have user stories, unit tests, and defects related to FEN string generation, algebraic notation, and displaying the chessboard in certain states.

Architectural Decisions

We decided to do MVC as our software design pattern. We realized when trying to explain this, that we were already somewhat doing MVC once we refactored our code a little. We decided that our Model is the ChessBoard class, which represents the chessboard as a whole. The Controller is `LaboonChessDocumentController.java`, which gives the commands to our View, which is the GUI itself and tell it what to display. The controller talks to our Model and requests information. There is a diagram of this below. We have decided to group our classes/objects into corresponding packages to make our code more readable. Objects go into the entities package, and services such as Stockfish go into the services package.

LaboonChess MVC Diagram


MODEL
(ChessBoard class)
STOCKFISH
VIEW
(GUI)
CONTROLLER
(LaboonChessDocumentController.java)