**LABOON-CHESS**

Charlie Mietzner (Scrum Master) -- clm133

Ritika Maknoor -- ritikamak

Peter Stamos -- petestamos

Rohan Patel --patelrohanv

1 December 2016

**CS 1530 – SPRINT 5 DELIVERABLE**

**DESCRIPTION OF SPRINT**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We went into this final sprint with a strong idea of what we wanted/needed to complete during it in order to meet a certain level of expectations that both our customer, as well as our own development team, had for our final product. By this 5th sprint, we, as a team, had a solid understanding of what each of our strengths were and what we would most likely be capable of accomplishing in the final two-week span. We started off this semester-long project with a strong and confident first sprint and subsequent finished-product delivery at the end of it. Therefore, we kept up that level of momentum throughout the rest of the sprints. By this final sprint, we felt confident enough that even when we felt as though we were in an unbearable time-crunch, we did not resort to cowboy coding and doing anything we could to get as much accomplished as possible. We stuck to our agile scrum mindset and continued to regularly/openly discuss what we were each working on with one another and were always sure to review one another’s code before pushing to our master branch.

During our previous sprint’s retrospective, we noted what our strengths and weaknesses were- both individual and as a group. We noticed that during our last sprint, we had very effective communication but wanted to improve on being more focused on implementing functionalities that our user noted as being high priority. Last sprint we did not complete implementing 2 of the 4 high priority user stories- ability to move chess pieces legally, and having turns be taken between player and computer. We reflected on where we might have wasted time on things that were of lesser priority. We noted that often times during the last sprint, time was wasted by multiple team members working on similar refactors of same functionalities and methods. Therefore, we decided that we needed to improve on being more strategic in how we divided up the user stories for this last sprint. We knew that we could not afford to have so many high priority user stories go not completed. We did not want to disappoint our customer yet again!

Our sprint planning session for this final sprint was effective and efficient. We discussed what we had completed so far- what components our “working software” had, and how close it was to meeting the end goal/all of the requirements for this product. During our first sprint, we broke down our backlog of user stories into 3 categories (Chess GUI; Game Mechanics; Backend). By the start of this final sprint, we had completed a large percentage of user stories from the Chess GUI category. We had the interface of the application (visual display and channel for user interaction) almost completely set up. We then discussed what the most logical approach should be for this final sprint. We went through our updated backlog of remaining/ incomplete user stories and chose our stories for the sprint. This time, however, for the first time in all of our sprint planning sessions, we ranked the chosen stories in terms of priority, in addition to story points. We divided them up amongst us, taking into consideration what domains and aspects of the development needed that each of us felt we were strongest and best fit to take on.

Also during this sprint, we interacted with our user. We were unsure of how important a few user stories were in relation to other more seemingly pertinent ones. In discussing with our customer, we realized that functionalities such as having images of each players taken pieces be visible on a side panel, were less important. While other user stories that we thought could be ignored if time did not permit us, such as having alert messages pop-up on the screen for illegal move attempts, were more important.

For this final sprint, we focused on completing the remaining requirements that dealt with core gameplay functionality. Up until this point, we had much of the game mechanics being worked on in separate branches that each addressed individual requirements such as ensuring the chess pieces move correctly and that the computer can react and make decisions based upon interpretation by the stockfish engine. This sprint would finally result in a fully functional game to deliver to our customer. In addition to the previous user stories that were to be implemented, was a kibitzer feature that would continuously generate random phrases to help train the player in how to better perform while distractions are in place. These final implementations proved to be difficult, but our group was able to confront these difficulties and deliver functional software to the customer.

Our work began by providing the stockfish engine with a means of communicating with our Java program. Once we began to bridge that gap in implementation we were able to move forward with development and start working out the primary user-facing mechanics. Specific chess functionality such as castling, pawn promotion, and enPassant were then implemented. At this point, we had to decide how we would approach the final days of the sprint and determine what features that were of highest priority that we could realistically have implemented by the deadline of the deliverable.

Our next hurdle was addressing the kibitzer implementation. Rather than modifying the GUI interface to accommodate for an additional field for the kibitzer output, we decided to generate the output into a separate window that would initialize when the chess game was opened. Once the kibitzer was implemented, we began to address the final core mechanics of the game. Now that the piece movement functionality was implemented and the stockfish engine was wired to begin making calculated decisions, we began to translate those mechanics into actual physical movements on the chess board. After numerous attempts and short-comings, we got the game to finally display movement and interact properly based upon the users play-style decisions. Now that the core game mechanics were in place, we were able to make small adjustments to the interface to accommodate for displaying which pieces were out of play and pop-up notifications to help the user understand what moves are illegal when they attempt to perform them. Our final implementation was providing the game with the ability to save and load their current or previous game session.

At this point, we completed the primary functionality of the chess application and were ready to deliver the software to the customer. Many of the user stories that we initially sought out to address were implemented and followed the initial specifications that were provided to us at the beginning of the semester. Although we were unable to address all of the user stories, we were able to address the vast majority of them, and more importantly, the ones that our customer highlighted as being of highest priority to them. Our greatest drawback was an overall lack of time. Had we had more time to implement these features, we feel confident that we would have been able to address the remaining user stories as well. Our group worked well and communicated efficiently throughout the duration of the software life-cycle. We had many goals from the onset of our development and persistently worked towards them to deliver a functional chess application.

**LISTING OF COMPLETED USER STORIES & STORY POINTS**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUBSYSTEM 1 – CHESS GUI:

* As a player

I want an automated kibitzer,

So that I can pretend I am in a noisy environment.

STORY POINTS: 8

* As a player

I want to first click a piece, then click its destination square, and then see the piece move to the indicated destination

           So that I can visually see that a move has been made.

STORY POINTS: 2

* As a player

I want for all pieces currently out of play for each player, to be listed on the side panel next to the game board

     So that I can easily assess the progress of each game.

STORY POINTS: 4

* As a player

I want the title screen of the game to read “LaboonChess”

           So that I know that the correct program has been opened.

STORY POINTS: 1

SUBSYSTEM 2 – GAME MECHANICS:

* As a player

I want a chess game that follows FIDE regulations for Standard American Chess

So that I have an accurate chess simulator that abides by national tournament standards.

STORY POINTS: 8

* As a player

I want turns to be taken (white/black/white/etc.),

So that the game follows standard chess rules.

STORY POINTS: 4

* As a player

I want an error message to appear if I try to make an illegal move

So that illegal moves are blocked from occurring.

STORY POINTS: 4

* As a player

I want special movements to be legal- such as “en passant”, “promotion”, and “castling”

So that the game follows standard chess rules.

STORY POINTS: 8

* As a player

I want it to be apparently visible when a king is in check

So that I can see if a player could potentially lose.

STORY POINTS: 2

SUBSYSTEM 3 – BACKEND:

* As a player

I want to be able to save and load games

So that I can pause a game and continue playing where I had left off, whenever I desire.

STORY POINTS: 8

* As a player

I want to be prompted with the option to start a new game or load a previous one; with the default being to start a new game

So that I can go directly to a new game if I have no old ones saved.

STORY POINTS: 4

* As a player

I want to save the game as a text file in .pgn format (portable game notation)

So that I can copy and load the file elsewhere if I decide to play on another machine.

STORY POINTS: 8

* As a player

I want to check if the .pgn file is valid

So that the program does not throw exceptions when trying to read from it.

STORY POINTS: 1

TOTAL VELOCITY = 62

**LINK TO CODE ON GITHUB**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

On GitHub: https://github.com/clm133/CS1530groupproject

**LISTING OF DEFECTS**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

The Stockfish engine would crash whenever opened.

DESCRIPTION:

The Stockfish engine adds functionality to the program by generating the best possible move for the computer to make.

REPRODUCTION STEPS:

1. Run “gradle test”
2. Observe failing test for StockfishTest.java

EXPECTED BEHAVIOR:

The test should not fail; the engine should start properly.

OBSERVED BEHAVIOR:

The engine would not start properly and thus was unusable.

NOTES:

This defect was fixed by editing the implementation of startEngine to first check what operating system the program was being run on. Then, the correct version of the engine was launched for that system. After this change, the defect was fixed.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

The program can export data to a file but it cannot properly import data and populate the board.

DESCRIPTION:

When trying to load a saved game, the program can read in the file, but it cannot populate a graphical board with the given information.

REPRODUCTION STEPS:

1. Run “gradle run”
2. Attempt to click load game and resume a saved game

EXPECTED BEHAVIOR:

The program should load the saved game and populate the graphical chess board with all of the appropriate pieces in their exact old positions.

OBSERVED BEHAVIOR:

The board does not populate with the old positions; stays in “New Game” or current game’s format.

NOTES:

This defect was the result of a discrepancy between the output that the Stockfish engine gives back after making a move, and the input that it requires to make a move. A workaround for this problem was reached- where every move that was made to get up to the saved point is displayed on the terminal. While working off these individual saved points, we were able to properly edit our functions to implement a fully functioning load game that did in fact populate the graphical chess board appropriately.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

The program can save a game partway through a game, but it cannot perform multiple tasks- cannot consecutively save a game and then also load a new game.

DESCRIPTION:

If one attempts to save a game and load a game consecutively in the same session, the program will throw an java.io.IOException: Broken pipe in addition to several more exceptions being thrown in JCL classes.

REPRODUCTION STEPS:

1. Run “gradle run”
2. Attempt to save the game; click “Save Game” button
3. Attempt to load a game; click “Load Game” button

EXPECTED BEHAVIOR:

The program should be able to save and load multiple times during the same session, without having to exit and restart the program between each action execution.

OBSERVED BEHAVIOR:

Program can only properly execute one action- save or load, per session.

NOTES:

This defect unfortunately cannot be fixed at this moment. However, we were able to recover from the error and allow the game to continue running without crashing. For the remainder of the session though, no saves or loads can be made.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

The program can identify a “check” condition properly, but it cannot identify any “checkmate” conditions.

DESCRIPTION:

During a game, when a “checkmate” is reached, every attempt at a next move is rejected because the king is in a “check” condition no matter where they move (the program does not allow a player to deliberately move their pieces in a way that will result in immediate subsequent death of their king piece). The program, however, has no means of informing the user that “checkmate” has been reached.

REPRODUCTION STEPS:

1. Run “gradle run”
2. Reach a checkmate condition

EXPECTED BEHAVIOR:

The program should inform the user that either they or the computer are in “checkmate” and thus that the game is over.

OBSERVED BEHAVIOR:

Every attempted move is rejected and the user is forced to infer that checkmate has been reached.

NOTES:

We attempted to correct this defect by using the Stockfish engine to generate all legal moves. This, however, proved to be a poor approach as we were not able to modify our system to accept the output that Stockfish provided us with.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

Colors for pieces were stored as booleans.

DESCRIPTION:

The representation of piece colors as booleans allowed for a simple implementation, but it also led to technical debt during later sprints.

REPRODUCTION STEPS:

1. Referring to source code

EXPECTED BEHAVIOR:

N/A

OBSERVED BEHAVIOR:

N/A

NOTES:

We had originally written our colors as booleans to easily keep track of black and white pieces. This, however, proved to be a poor design decision. As our program grew more complex, it became difficult to keep track of several other colors/boolean variables. Looking back, it would have made more sense to use an Enum or separate color class.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

Several constant values are duplicated repeatedly throughout the various classes.

DESCRIPTION:

There exists a large number of variables that are final constants. These constants are the same and stay unchanged.

REPRODUCTION STEPS:

1. Referring to source code

EXPECTED BEHAVIOR:

N/A

OBSERVED BEHAVIOR:

N/A

NOTES:

This defect was a result of incorrect planning when designing classes. Many constant values are redefined across several classes. It would have made more logical sense to keep a static class of constants that would be used consistently across the various classes in our program.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SUMMARY:

When running test cases, the PlayerTest.java test file detects a defect where a popup alert message temporarily blocks the running program.

DESCRIPTION:

The program will not continue to test or build until “OK” button is clicked on the popup alert window.

REPRODUCTION STEPS:

1. Run “gradle test”

EXPECTED BEHAVIOR:

The tests should run without any interruption.

OBSERVED BEHAVIOR:

The tests cannot continue to run until “OK” button is clicked.

NOTES:

This defect is related to our implementation of checking for illegal moves and checking if a player’s king is in check. By constantly checking for these conditions throughout the entire run of the program, we have resulted in error messages occurring before we even want them to be checked for. Due to this incorrect implementation, we were unable to change the way that the tests function without first changing the methods that we were testing to no longer output warnings as popups.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_