**CHESS APPLICATION**

Charlie Mietzner -- clm133

Ritika Maknoor -- ritikamak

Peter Stamos (Scrum Master) -- petestamos

Rohan Patel --patelrohanv

27 October 2016

**CS 1530 – SPRINT 3 DELIVERABLE**

**DESCRIPTION OF SPRINT**

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

We began this sprint with a strong idea and grasp on what we wanted to complete during it. By this 3rd sprint, as a team, we had gained a solid understanding of how the 2-week process ahead of us would most likely carry out. We had a general plan for what we would complete in the rest of the sprints that we had ahead of us, and therefore were more confident throughout this sprint, in comparison to the previous ones, about *how* exactly our work during it would lead to the completion of our final product. 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 decisive/clear on what we wanted to accomplish by the end of the sprint. We wanted for all of the teammates to have a good understanding of what our principle goal was for the sprint, and how each user story assigned to that sprint would contribute to the accomplishment of that principle goal.

Our sprint planning session for this 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), and had a vague idea of how we felt would be more logical to go about completing all of them. During our second sprint, using that initial backlog with the categories, we chose and completed user stories that were centered around setting up our Chess GUI and determining our graphics. Once we went over what we had completed, we discussed what that the most logical next step/principle goal for this sprint would be- to have our backend of game mechanics be all set up. We decided that by completing that this sprint, in the next sprint we would be able to synchronize the GUI and backend of game mechanics and finally have a Chess GUI that the user could begin to interact with by moving pieces around the board. Having this principle goal determined, we were able to choose our user stories rather quickly and divided them up amongst us by taking into consideration what domains and aspects of programming each of us felt we were strongest and most familiar with.

Throughout the entirety of the two-week span, we communicated very well. Due to the fact that the user stories that we chose to complete were all very much intertwined with one another, we were all in almost constant contact. The nature of the tasks that the user stories called for required manipulating one another’s code- i.e. abstracting away methods, organizing classes, synchronizing objects, etc. We all actively communicated and were available whenever another teammate needed anything. We did not have many disagreements. The only issue that arose was the fact that so many of our user stories overlapped in terms of the code we were working on. It was difficult when multiple team members were working on the same classes- such as a specific piece class and both members were re-organizing the code/methods in different ways. However, we resolved this for the most part pretty well by communicating with each other about who would modify what they needed to first and who would do so second. We were also particularly active with often merging our individual completed modifications to the master branch. By doing so, we all always had the most up-to-date version of the code subsets. After about the first week of the sprint, we, as a team, had a cohesive system for working on the code efficiently.

We did not interact with the customer during this sprint. After the previous sprints, we had a Chess GUI set up and most of the graphics determined. After this third/current sprint, we have finally completed setting up our backend of game mechanics (classes for board and piece movements that deal with the actual algorithmic implementations of the rules of chess). At the end of this third sprint we do indeed have completed and functional “working software”; however, this version of the software does not yet allow the user to interact with the board/Chess GUI. During our next sprint we plan to synchronize the GUI and the backend, and therefore during our next sprint (after we begin to synchronize the two), we plan to consult our customer and show them our interactive graphical display. We hope that doing so at that point in the near future, rather than now, will allow the customer to finally interact with the GUI physically. We will perform user observations as we take note of how they interact with the system by default. We hope that at that point, their feedback and our observations will provide us with a sense of direction- whether our understanding of how they expect to perform tasks (such as moving chess pieces on the board) is in line with theirs, or not. By having customer interaction occur near the beginning of the next sprint, we hope to solidify our understanding of the customer’s expectations before spending the entirety of the next sprint writing invalid software.

We faced many challenges when writing our code. During this sprint, we were focused on the backend of game mechanics. We already had created our graphical product and developed components of it, but now we had to actually write code that dealt with the algorithmic implementation of the rules of chess. This was not an easy task because our program had to be dynamic enough to handle user interaction, GUI interaction, and “computer opponent” interaction. It was not simply a program that would run once, take in user input for moves, and then complete. We began by writing a skeleton backbone for what our code would have to encompass- classes for different pieces and for the board itself. Then, we decided that abstraction was necessary. We created an abstract class Piece that was a generic Piece object; therefore, as we developed further, we could use it as a superclass to more specific piece types: kings, queens, knights, bishops, rooks, pawns. We decided that this was the most effective way to go about writing clean and useable object-oriented code. This helped us have uniformity amongst all of our pieces; we simply had to then write more specified code for each type of piece in terms of how exactly they could move on the board. Once we began to brainstorm how to effectively organize our code, we became much more familiar with what seemed to work best for us and would minimize duplication of code. We did not come across any challenged when writing our tests once we got started and figured out what exactly we needed to test for- both base cases (expected use cases) and edge cases (unexpected use cases). It was nice that we could write similar tests for the different piece types, since we were testing similar actions- valid/invalid movements, bounds on board, and occupancy of same/opposite pieces in squares.

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

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

* As a player

I want to play against a computer

So that I can simulate playing against a person.

STORY POINTS: 2

* As a player

           I want the game to run on the JVM

           So that it can be portable.

STORY POINTS: 1

* As a player

           I want the application to run on a 512mb machine

           So that it can run on a variety of systems.

STORY POINTS: 1

* 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: 4

* As a player

I want the pieces for each player to be set-up in their correct position/squares on the board at the start of a game

So that the game follows standard chess rules.

STORY POINTS: 4

* As a player

I want the board to be organized so that no 2 pieces can ever occupy the same square

So that the game follows standard chess rules.

STORY POINTS: 4 **(not completed)**

* 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 **(not completed)**

* As a player

I want pieces to not make illegal moves

So that neither player can cheat.

STORY POINTS: 16

* As a player

I want the game to prevent any pieces from moving outside of their range/path

So that the game follows standard chess rules.

STORY POINTS: 16

* As a player

I want synchronized steps for movement for all pieces

So that the game is smooth and no pieces require special clicks and such.

STORY POINTS: 8

TOTAL VELOCIY = 52

**LINK TO CODE ON GITHUB**

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

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

**DETAILS OF WHY USER STORIES WERE CHOSEN**

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

When we had our initial discussions with the customer, they had provided us with a list of requirements that they had in mind for the software. After going back and forth with them- asking questions and providing feedback, we were able to iron out a cohesive list of details of what the final product was expected to be. We then divided our backlog into 3 categories- Chess GUI, Game Mechanics, and Backend. When we first created our backlog and began to develop these 3 categories/subsystems, we had discussed as a group that perhaps the best way to complete the final product would be to complete a handful of user stories from each of the 3 categories, during each sprint. We believed that approaching it in this manner would ensure that we did not spend too much time/too many sprints on certain subsystems and neglect others.

During this sprint’s planning session, we decided that our main goal at the end of this sprint would be to have our backend of game mechanics be all set up- meaning completing all requirements that deal with the actual algorithmic implementation of the rules and restrictions of chess. By setting up our backend, we will be able to focus on finally synchronizing the backend and the GUI during the next sprint. We successfully completed implementing most of the backend for piece movements and game mechanics/policies/set-up.

We decided on 10 user stories in the end- a list comprising of both original-backlog user stories, and modified/added user stories. Chose original-backlog stories that were not too specific in relation to the backend; chose stories that would help us with initial set-up of the backend so that we could further develop it to meet more detailed and specific user stories later. We added stories by first picking out user stories from our original backlog that seemed to be broad and overarching for the software- such as stories about how pieces should move and be able to be manipulated on the board. We then divided these such stories into more specific and detailed user stories that could help us, as a team, better divide up the tasks at hand.

-----In original backlog; SUBSYSTEM 3-BACKEND-----

* As a player

I want to play against a computer

So that I can simulate playing against a person.

(We simulated this when we set up the game; we placed every single piece on the computer side in the correct place. Therefore, if we pick white, computer will be aligned as black pieces.)

* As a player

I want the game to run on the JVM

So that it can be portable.

(Wrote it in java and built it using gradle.)

* As a player

I want the application to run on a 512mb machine

So that it can run on a variety of systems.

(Runs off the JVM which can run on this condition.)

-----In original backlog; 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.

-----Added stories; SUBSYSTEM 2-GAME MECHANICS-----

* As a player

I want the pieces for each player to be set-up in their correct position/squares on the board at the start of a game

So that the game follows standard chess rules.

(This is in backend- in player class; puts every piece appropriately in correct spot.)

* As a player

I want the board to be organized so that no 2 pieces can ever occupy the same square

So that the game follows standard chess rules.

(In movePiece method. NOT COMPLETED- Only done for certain pieces. We are moving pieces via the controller but the work is done in backend; so sending info back and forth between them, which is why we haven't finished this. Is high priority for next sprint.)

* 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.

(NOT COMPLETED- Wanted to first complete the basics in terms of backend- getting all piece classes done. This requires knowledge between backend and the controller also. Moved to next sprint.)

-----Added stories; SUBSYSTEM 2-GAME MECHANICS; **added** **for refactoring purpose**-----

These are all part of refactoring; all deal with making code better/simplifying the piece class and making it modular- individual modules to follow OOD. Completed these user stories by using abstract classes for piece.

* As a player

I want pieces to not make illegal moves

So that neither player can cheat.

* As a player

I want the game to prevent any pieces from moving outside of their range/path

So that the game follows standard chess rules.

* As a player

I want synchronized steps for movement for all pieces

So that the game is smooth and no pieces require special clicks and such.

**LISTING OF DEFECTS**

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

SUMMARY:

The bishop class currently cannot check if its movement path is impeded.

DESCRIPTION:

When observing the bishop class’ implementation of movement, does not check if there is a piece that impedes the path between the current position and destination.

REPRODUCTION STEPS:

1. Refer to source code for the Bishop class.

2. Inspect the implementation for movePiece method.  
EXPECTED BEHAVIOR:

The method should return false if the path is impeded.

OBSERVED BEHAVIOR:

The method returns true if the destination is valid according to a bishop’s legal moves, not if it is valid in accordance to the board.

NOTES:

This sprint we focused on implementing movement for all pieces and setting up a relationship between all piece classes; we agreed to focus on the movement in regard to game mechanics in the next sprint. Will address this defect further then.

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

SUMMARY:

The rook class currently cannot check if its movement path is impeded.

DESCRIPTION:

When observing the rook class’ implementation of movement, does not check if there is a piece that impedes the path between the current position and destination.

REPRODUCTION STEPS:

1. Refer to source code for the Rook class.

2. Inspect the implementation for movePiece method.  
EXPECTED BEHAVIOR:

The method should return false if the path is impeded.

OBSERVED BEHAVIOR:

The method returns true if the destination is valid according to a rook’s legal moves, not if it is valid in accordance to the board.

NOTES:

This sprint focused on implementing movement for all pieces and setting up a relationship between all piece classes; we agreed to focus on the movement in regard to game mechanics in the next sprint. Will address this defect further then.

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

SUMMARY:

The queen class currently cannot check if its movement path is impeded.

DESCRIPTION:

When observing the queen class’ implementation of movement, does not check if there is a piece that impedes the path between the current position and destination.

REPRODUCTION STEPS:

1. Refer to source code for the Queen class.

2. Inspect the implementation for movePiece method.  
EXPECTED BEHAVIOR:

The method should return false if the path is impeded.

OBSERVED BEHAVIOR:

The method returns true if the destination is valid according to a queen legal moves, not if it is valid in accordance to the board.

NOTES:

This sprint focused on implementing movement for all pieces and setting up a relationship between all piece classes; we agreed to focus on the movement in regard to game mechanics in the next sprint. Will address this defect further then.

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

SUMMARY:

The pawn class currently cannot check if its movement path is impeded.

DESCRIPTION:

The pawn has an implementation of capture that incorporates capturing pieces in occupied squares. However, it does not check if the square it is moving to vertically is currently occupied or not.

REPRODUCTION STEPS:

1. Refer to source code for the Pawn class.

2. Inspect the implementation for movePiece method.  
EXPECTED BEHAVIOR:

The method should return false if the path is impeded.

OBSERVED BEHAVIOR:

The method returns true if the destination is valid according to a pawn legal moves. It should return false if the move is illegal or square is occupied (unless it is a capture).

NOTES:

This sprint focused on implementing movement for all pieces and setting up a relationship between all piece classes; we agreed to focus on the movement in regard to game mechanics in the next sprint. Will address this defect further then.

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

SUMMARY:

The pawn class currently cannot check if its captured piece is an enemy.

DESCRIPTION:

The pawn has an implementation of capture that incorporates capturing pieces. However, it does not check if the square it is moving to is currently occupied by a piece of the opposite player’s color.

REPRODUCTION STEPS:

1. Refer to source code for the Pawn class.

2. Inspect the implementation for movePiece method.  
EXPECTED BEHAVIOR:

The method should return false if the piece to be captured is of the same color.

OBSERVED BEHAVIOR:

The method returns true if the piece being captured is either of the opposing color or of the same color. Returns true for both cases.

NOTES:

This defect was corrected by making changes the Piece class’ implementation of movement- where it checks the piece in the square it wants to capture. Checks the color of the piece in currently occupying the square. If the piece is of the same color, it will now return false; else, it will return true.

**DETAILS OF ARCHITECTURAL DECISIONS**

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

Under its current architecture, the board is split up into an 8 by 8 grid of squares. This is implemented as a two dimensional array so that it would have quick and easy to the square at a particular set of coordinates. The square class itself keeps track of its current location (which on the board will not change) and the piece that is currently at that location. Individual pieces are extended from an abstract class. This allows us to store each player’s pieces in a list of generic pieces, instead of simply as objects- which would be the case if each piece was instead just an entirely separate class.

Each piece type has a unique/uniformed move method. Using abstract superclass for piece and having each piece type have the same methods, we are able to keep this relationship intact. It allows us for there to be a relationship between all the pieces, while also allowing for a more specified relationship between pieces that are all of the same type. For example, it allows for a rook to move like a rook, a bishop to move like a bishop, etc. This proved to be a better choice than the alternative of having one large piece class that when clicked, would first have to check what type of piece it is and then figure out how to move it depending on its type.

This will also come in use later on once we implement pawn promotion. When a pawn reaches the end of the board, it can turn into any piece that it wants to (any type except for king). This, now allows us to easily reassign a pawn to a higher piece when it can be promoted, rather than us searching for and removing that pawn from the list and adding a new piece to the list in its place.

We decided to add 3 items to our backlog to potentially refactor some of our code. All of the added user stories deal with making our code better/simplifying the piece class and making it modular- individual modules to follow object-oriented design patterns. We completed these user stories by using abstract classes for piece.

* As a player

I want pieces to not make illegal moves

So that neither player can cheat.

* As a player

I want the game to prevent any pieces from moving outside of their range/path

So that the game follows standard chess rules.

* As a player

I want synchronized steps for movement for all pieces

So that the game is smooth and no pieces require special clicks and such.

We grouped our classes/objects into 2 major subsystems- piece and board. We technically also have a 3rd subsystem for the GUI which only represents the backend- the idle state/visual representation of the board. We later want to implement another subsystem that handles the actual user input- could be “controller” for example. This new subsystem would provide us communication between the backend and the GUI.