RoboChess

PRELIMINARY DOCUMENT – CONSTRUCTION EXPERIENCE
VERSION 2.0
NOVEMBER 27, 2016

Prepared by: Team B5
LIN, YUCHEN | NELSON, JORDAN | PARK, RYAN | WANG, XINGENG | VAN HEERDE, WILLIE

Table of Contents

1. Pair Programming Sessions

1.1 Pilot – Jordan Nelson

1.1.1 Co-Pilot – Ryan Park

My first pair programming session was with Ryan. During our session, we implemented the game board's coordinate matrix transformation system. This wasn't an especially difficult problem, but one which would have taken much longer to do individually. While the pilot programmed, the co-pilot worked to draw out hex grids and coordinates to develop and understand with the pilot algorithms to transform each coordinate depending on the direction faced. Once a direction was figured out, the pilot would then add it to the algorithm. I found this session extremely enlightening, as I estimate personally implementing a similar piece of code would have taken me at least twice as long, but probably much longer.

1.1.2 Co-Pilot – Willie Van Heerde

During my pair programming session with Willie, we implemented the ReferenceSheet class, which keeps track of the unchanging values for each robot (such as range, damage, and movement range). We picked this class not because it was especially difficult, but because in lecture it was mentioned that some companies operate with a standard pair programming policy, and I believed there was value to be had in seeing how pair programming faired when implementing a relatively simple class. That being said, it was useful with our implementation of the ReferenceSheet class to have a co-pilot always checking the pilots' values. Without going into an overly large amount of detail, our implementation had each robot type be indexed by an integer (from 0 to 2) and so it seemed natural to hold the unchanging values in small arrays indexed by the corresponding robot type (so if a tank had an index of 1 and you want to know the range a tank has looking up the value would be as simple as "robotRange[1]"). This being the case, the co-pilot was useful to check that the values were being stored in the array in proper order, as they were very easy to mix up.

1.2 Pilot – Yuchen Lin

1.2.1 Co-Pilot – Ryan Park

During our session, Ryan and I developed the Server class together. During my turn as a pilot, I wrote the helper functions that connected the coordinate transformation system of the game board which was implemented by Jordan and Ryan. Ryan helped me with building the communication system using Akka between the server and player because Ryan is more familiar with it than me. During my turn as a co-pilot, Ryan manipulated the message passing between the Server and Player(client), and the player queueing system which is used for managing turns. Having the pair programming session was good for avoiding mistakes and allowing each person to focus more on the task he is good at. Also, having someone asking

questions about the code at the same time is good for unscrambling the thoughts of the programmer.

1.2.2 Co-Pilot – Xingeng Wang

During our session, Xingeng and I worked together to continue implementing the Server class. During my turn as a pilot, I wrote the helper function that manages each turn. And Xingeng helped me to connect the manage turn function with the part he wrote, which was the move function. During my turn as a co-pilot, I help Xingeng with the move function. Since the move function needs to manage the turn when the previous player ends their turn, getNextRobot and getNextPlayer are useful for the move function. With pair programming, it saved time to read the other person's code to use it. Not only avoiding mistakes, two programmers splitting an important function can also save more time than one person writing it all.

1.3 Pilot - Willie Van Heerde

1.3.1 Co-Pilot – Xingeng Wang

For my first session, I paired with Xingeng and we chose to work on the playerInfo class. Xingeng started first with programming and as he was coding I peer reviewed his code as he was working on it while making some tests for the static main class. Because I had implemented the robot class by myself I knew what procedures should be done when creating the three robots in the playerInfo constructor. I used this knowledge when we switched roles in our pair programming session to ensure the test cases we were both working on now would work. After I had finished coding we had to make minor changes to the code from the few bugs we found when using our test cases. Because of the simple nature of the class we were both able to quickly and efficiently implement the class that was integral to the server class that Xingeng was working on as well.

1.3.2 Co-Pilot – Jordan Nelson

For my second session, I paired with Jordan and we choose to work on the reference sheet class because it is a simple to implement class that is essential to the project to function. Because this class was relatively simple it was easy to make an error because it was so simple there was little planning before implementing the code. For example, I was fixated on using a linked list to store the value for each of the statistics but, Jordan pointed out that this was unnecessary because the values never get removed, altered, or reordered so a simple array would work (0 for scout, 1 for sniper, 2 for tank). When we finished our session we both had made test cases that ensured when we did our final walkthrough we would know if there were any bugs in our code. Because of this we were able to quickly finish the class and implement tests in the static main that verified that each thing works as it should. By working together in our pair programming session, we were able to quickly finish and test the work we sought out to do.

1.4 Pilot – Ryan Park

1.4.1 Co-Pilot – Jordan Nelson

During our pair programming session, Jordan and I co-implemented the game board's coordinate transformation system. While one of us was coding, the other would draw pictures of the hexes and their corresponding mappings. The co-pilot would also be responsible for developing the arithmetic algorithms on paper while the pilot was responsible for translating this algorithm to code. Having this separation of concern allowed both pilot and co-pilot to focus on their respective tasks.

1.4.2 Co-Pilot – Yuchen Lin

During our session, Yuchen and I continued developing the Server class. During my turn as the pilot, I implemented message passing between the server/player as well the game's player queueing system. Since I was not concerned with handling the player actions, I could quickly outline the servers main structuring. While I was the co-pilot, Yuchen wrote the various helper functions that would utilize the coordinate transformation systems (developed by Jordan and I) to handle the manipulations of the game board.

1.5 Pilot – Xingeng Wang

1.5.1 Co-Pilot – Willie Van Heerde

During our pair programming session, Willie and I co-implemented the PlayerInfo class which is not one of our major classes so its size is not very big or hard to implement. Because of this reason, I took the role of pilot on implementing all of the functions in this class but the test. When one of us was coding, another one was always very helpful to check if we made a mistake and if we followed the coding style properly. When I was in the co-pilot role, I helped Willie check if there was some test case which might happen but we didn't test yet so we can save time in future. Willie also fixed some implementation that we missed that we did not discover until we began testing.

1.5.2 Co-Pilot – Yuchen Lin

Yuchen and I did pair programming on part of the server class, which has functions that are not easy and normally not small in size. It is very easy to make some mistakes for these reasons. With another pair of eyes on the screen, the mistakes we made were much easier to find. Before we began coding, we always discussed the way we could implement each function and to see if there were better ways to implement them, and we felt this was helpful for us. We switched our role every half hour to 45 minutes so our brain and hands could take a break.

2. Code Review

2.1 Code Selected – Interpreter Class

For our code review we chose to examine the Interpreter class, we did this because it is among the most complicated components of the project. Ryan Park did almost all the implementation because the rest of the group was unsure of how to implement the class and had agreed to implement other classes. This makes it a prime candidate for our code review because since the class has been implemented our group can go through the code and voice any questions we have regarding the implementation. If the code has been properly formatted and uses proper conventions, we should have little to no questions about the implementation.

2.2 Good Points

The class was written very well in many regards, the most distinguishing feature was the proper formatting of the code and how it followed the designated coding convention. Another key distinction was that all important fields and methods had an accompanying JavaDoc that explained the field or method concisely within one sentence. By doing so the code is very easy to understand as there is no needless filler that convolutes and snowballs out of control in the class. Finally, all the variables and functions were given descriptive and concise names that perfectly described what their purpose was and what it did. With both the proper naming and JavaDocs the group had very few questions even though the actual class was quite large and foreign to most of the group. Overall the Interpreter class was implemented very well with proper conventions followed at all times, which resulted in an implementation that was both easy to read and understand for one of the most complex components of the project.

2.3 Bad Points

There were only 2 aspects of the class we felt that could be considered undesirable in the implementation. The first being that there was a massive switch case that was responsible for the forth interpreter. This has roughly 40 cases and is slightly unwieldy to read but nothing can be done as it is not a formatting error; rather the only means to implement it. The second negative aspect was the class could be optimized in a few key points such as the fact that it performs 2 passes through the script which could be reworked into a more efficient 1 pass implementation. Some of the data structures could also be reworked such as using a regular map instead of a bidirectional map. Overall there was very little negative points because the few things that were there did not effect the project's runtime adversely.

2.4 Overall Group Opinion/Discussion

The group was extremely pleased with how well the class was implemented and how it was broken down into such small well documented sub problems. The interpreter, which was once

a mysterious and unknown part of the project, could almost be read as a book with how smoothly everything was explained.

As stated in the previous section the group found very little wrong with how the class had been implemented. There are two primary things within the Forth interpreter we would like to change. The first is the massive switch statement, which can not really be changed as it relies on checking which of the 40 possible actions it should take. The second thing is that the class could be optimized slightly so that it uses a one pass through system and changing some of the data structures to less resource demanding variants. Ultimately both of these could be altered but this would be done if we have enough time to do so as it would be a considerable undertaking with very little to gain from.

3. Changes

- Several extra classes were needed in the GUI. For starters, we needed several extra classes to handle the create game sub menu system. This wasn't originally planed for in our design document. We also decided to add a class for both the server class and the player class that handles the GUI system for both classes. These classes are named serverGUIHandler and playerGUIHandler respectively.

- Within the server class, we needed to add the 'onRecieve' function, which is an overridden Akka function that handles received messages. We also added several private helper functions that support server computation. These include checking if a tile is in range, the distance between two tiles, etc.

- Within the player class, we realized that all of the functions within the design document could simply be handled within the Akka onRecieve function. The isAI function got moved to the playerInfo class, which gives access to that function in other classes and keeps our implementation consistent with object-oriented programming.

- We also added another package with contains roughly twenty message classes, which are used in Akka to communicate between the client and the server within our architecture. These classes are minor, but a major part of how our architecture works, and something we missed within our design document.