RoboSport 370

Implementation Document

Group C3

Corey Hickson - crh208

Daniel Sanche - drs801

Landon Tetreault - lzt639

Levi Paradis - ldp343

Matthew Wind - mdw466

November 29th, 2015

I. Introduction

This document describes the implementation of our RoboSport370 project. It includes a git tag linked to our most-up-to date implementation as well as a reference to instructions on running the project. We discuss which coding style we used and include our personal reflections on the experience each of us underwent during the implementation. As well, we discuss implementation statistics and testing, including the coverage and methods we used. Next, we explain division of labour: who did what on this portion of the project. It will be slightly different than the commit log from our repository since we included pair programming, mob programming, and any other collaborations which resulted in only one commit for multiple people. Implementation completeness is discusses where we are currently at in our implementation, as well as a plan for what we will need to fix, add, and update in the future. Design deficiencies explains limitations in our current implementation. As-built design discussed how the implementation relates to our design document in a more abstract and high level language. Additionally, there is a user manual which includes screenshots of every user interface in our application and instructions on using the application. Lastly, we conclude our report and summarize our findings.

II. Git Tag

The url for our git repo is https://git.cs.usask.ca/370-15/370-15c3.git. The code created for the implementation has been tagged as “Assignment_3”. Instructions on how to run the program can be found in the README.md. Essentially, download the RoboSport370.zip file, extract and run the exectuable .jar.

III. Coding Style

Before starting the implementation, we discussed which coding style standards we would follow. We decided to use the Google Java Coding Style (https://github.com/google/styleguide).

Furthermore, we decided on the following coding standards:

- we will put opening braces on the same line as the if or loop statement,
- braces should be used even for single line statements,
- JavaDoc comments should be added to each method,
- variables will be written in camelcase, with the first character in lowercase,

- packages and classes should be titled using camelcase, with the first character being a capital letter,
- constants and enums should be used instead of “magic numbers,”
- and constants are written in all caps, with words separated by underscore.

Additionally, we discussed which git branching standards we wished to use. We decided to base our git branching model on Vincent Driessen’s git flow model: every time we wanted to add a new feature, we would create a branch called “feature/feature_name”, which branches off of the main development branch. After the feature is complete and it has been code reviewed, we would merge the branch. Our practices differed slightly from git flow however, because we decided to use the master branch as both to develop and keep the primary copy because there was no need to keep a pristine, production quality branch for the project.

Lastly, our group made sure to write code in such a way to minimize errors. For example, organizing cases from most important or common to least important or common. Code had to be compiled and tested with print statements before allowing pushing it to the repository, along with this we decided code must be tested and reviewed before it would be merged into the master branch. This allowed us to have a bug free repository and improved efficiency in the long run.

IV. Reflections

We used pair programming as much as possible when the coding was difficult. All of the group members found it to be very helpful. We found it reduced errors and we were much less prone to making small mistakes, even though it took a longer amount of time to produce the same amount of code. An important aspect of pair programming was the co-pilot’s ability to ask other team members for clarification about code specifics while the pilot did not have to stop their coding. This improved workflow greatly.

We did a couple code reviews throughout the implementation process to help with debugging and formatting, as we found having another person read our code was extremely valuable when it came to finding errors. Below is the notes from one of these reviews.

Notes from code review for controller package November 25th:

Code was reviewed by Levi Paradis and Matt Wind

The first section of our code review was based on the structure of the code. We checked the code implemented the design correctly, which it did. The code followed the Google Java coding standards correctly. It was well formatted, but was missing javadoc comments in places, which we inserted as we went through the review.

Some of the functions were unnecessarily big and clunky, these were either broken into smaller functions, trimmed down to be smaller, or cut completely if they were unneeded. The biggest issue with the structure of the code was the number of “magic numbers” in our code. This issue has not been fixed as there are a high number of instances where this is an issue, and will need to be fixed in the future. As stated above, some of the documentation needed to be fixed, although most of it was extremely well done. We noted missing Java Doc comments in places, and fixed these as we found them. We also encountered a few unneeded TODO comments, which were removed when we encountered them.

At the end of the review, the comments were consistent between every class in the controller package. Variable definitions were largely done very well. They all had clear, well defined names that explained what the variable was used for. There were no redundant variables in the controllers, every one of them had a specific purpose and was utilized efficiently.

Arithmetically the code was sound, it avoided most of the general problems such as comparing floating points for equality or forgetting to account for rounding errors. This would likely have to be checked again when the game controller is finished as it will involve some complex arithmetic involving path finding algorithms and such. We found that defensive programming was not used in some places, which will need to be fixed in the future to make the program more robust.

During our implementation we ran into a few different obstacles. The first one was with Git. Creating branches ended up being a difficulty. Initially, we often made big branches, but it was hard to keep them in sync and merge them together without conflicts. We realized soon after some of these conflicts that small branches and frequent commits helped minimize the issues.

Another obstacle we ran into, as mentioned above, was that git merge conflicts were difficult to solve. We eventually minimized these by using smaller branches and frequent commits, and by using new tools that make merge conflicts easier to resolve.

We found it very hard to make clean code for graphical user interface view files. There was a lot of copying and pasting big blocks of code, which made it hard to find anything useful in the classes. We also ended up with a lot of magic numbers to replace, because it was hard creating constants for every possible ui element position.

We found that the more important the class, the more likely it was to get messy. Lots of people were writing to it, so there were a lot of loose methods that nobody took responsibility for. For example, the game controller central to every part of the project.

It was difficult to enforce a code style as everyone has their own habits and tendencies. We fixed some of this by formatting the code using the Google Java code style before committing it.

We also found it hard not to add extra features. As we were making classes, we kept coming up with new ideas for how to make things better, but realistically we could not implement every idea, instead needing to focus on what we promised in the requirements document.

Working with a very visual project (i.e. a hexagonal grid with moving pieces), it was helpful to keep a written workbook during the project. This allowed for better understanding and conceptualizing of what was happening before writing a function which implemented the concept. This documentation can be seen in our Git repository under ‘doc>impl.’

Reflecting on our group’s structure, we have found we worked with a very flat group structure which aimed for substantive equality between the members opinions. With no designated leader making choices, this flat structure allowed for greatly discussion of ideas and possible solutions.

Something we learned about about software engineering was how helpful it can be to develop a project in the same room while working together. Almost all of our work was done together in a collaborative environment. There was always

open feedback with the other members when questions needed to be asked. We found this much better than dividing the work among all members and then ‘gluing’ the work together, as many of us have done in past projects. Overall working on the project together as a team created a much more efficient process and helped every member learn more in the process.

V. Implementation Statistics

From our implementation, we came up with a few statistics about what we achieved. They are as follows:

- files: 53
- lines: 7225
- classes: 57
- calls: 2132
- statements: 3853

VI. Tests

Due to time constraints, we decided against having full code testing coverage. Instead, we put tests in the main functions of our major classes. These tests create an object, call some methods, and verify the results. The main classes would also open up the graphical user interface (GUI) to allow testing of different views and their interactions with the controllers.

A reasonable portion of our GUI was tested through visual testing. In our reflections, this was not the most effective way to test a GUI. However, it was similar to what we expect might happen in a later stage quality assurance group.

Additionally, we used a lot of tests through post implementation tests in each of the classes. Each class was responsible for testing itself. Near the end, we aimed for a greater scale testing which linked all of the classes together to search out bugs which happened between the interactions of the various classes.

VII. Division of Labour

The division of labour in the team was shared between team members. Every team member had the opportunity to work on important pieces of code and each

member tested and reviewed code during down time. The specifics of each team members are as follows.

Daniel Sanche: Was responsible for the interpreters. Created the ForthInterpreter class and related classes for parsing the forth code into java function calls. Also wrote the JSONInterpreter for reading JSON files and converting them into Java classes. This class ended up being responsible for communicating with the robot librarian server as well, as it was written to communicate using JSON. Daniel was also responsible for creating the views and controllers used to manage, and edit robots, and add robots, because these classes communicated with the robot librarian. He also created the consolve view on the game screen, displaying forth actions as they are run. After getting some experience in working with the graphics libraries, he helped others implement other views in the system.

Matthew Wind: Created the end view and end controller. Worked on the robot picker view. Small work on game controller to implement end game. Helped create user manual.

Corey Hickson: Created the main menu view, setup view, and map view. Created art and acquired assets for the interfaces. Implemented portions of the setup and game controllers. This process included importing and managing the LibGDX library. Part of that management included making additional helper classes, such as an audible timeline for animations, hideable sprites, and a sprite accessor.

Landon Tetreault: Created and implemented Map, and Tile classes. I also implemented a large portion of the GameController class. I spent several hours figuring out an algorithm that properly created the tiles and set their positions using a map that I drew out. Corey and I worked together to create the getDirection functions in the Map class as well as moveRobot in the GameController. Daniel and I spent a number of hours debugging the entire program and getting it to finally compile and run the moves and call the view operationsproperly..

Levi Paradis: Integrated the setupController with the setupView class and the mainMenuView class. Integrated gameController class with the mapView class. Pair programmed the editTeamView class, along with the gameController and setupController. Reviewed and tested most of the classes before they were

merged to the master branch. Designed additional GUIs that weren't originally included in the requirements and design documents. Exported the executable .jar file to the git. Created the README, Changelog, User manual(with matt), and a large portion of the implementation document

VIII. Implementation Completeness

As of November 29th, 2015, our project is in early alpha stage of development. While it is functional and is able to fulfill some of the requirements of the customer it does not currently have all of the features that we would like to have. We predict we would need another 1-2 weeks of development to bring the product out of alpha and into a beta 1 release.

Features that are not in the alpha but will be added in the beta release:

- robot movement is buggy. It is possible for robots to move off the board in some cases
- the project does not communicate with the robot librarian server
- debug mode does not allow us to edit robot stats at run time, as we would have liked.

Features that are not in the alpha but will be added into the full release:

- map randomization; as of now the map is a pre-generated seed and will only change with map size,
- advanced game logging; more tools for logging game information each turn including a way for this information to be exported out of the program and allowing matches to be replayed using old log files,
- user interface polish: the current user interface is not very user friendly overall, in the full release there will be a more visually appealing user interface,
- and rewind capabilities: we would like to have the capability to rewind steps during a game.

During the development process we found many opportunities to improve upon our original design. We did this by adding new features or just making some thought out features more user friendly. Some of the improvements we added to the design are as follows:

- we added four different speed settings (1x, 2x, 4x, and 16x), with the possibility of extending many more,
- we have added a user interface for creating and editing teams and robots,

- the user interface for creating robots allows the user to create new robots as well as modify the stats and add Forth code,,
- this will add robots to the RobotLibrarian server
- we added the ability to pull robots from the RobotLibrarian server and display their information when making a team,
- and we added the ability to create and add custom robots with the interface.

The project was designed to be as reusable as possible, which means most of the code could be repackaged and used in other projects. While no plans to reuse the code exist yet this would be possible. Specifically with the interpreters: if another project were to require a Forth or JSON interpreter, reusing the code for that purpose would be easy and efficient. Lastly, the map, view and the model, would also be able to be reused very well.

Lastly, our implementation contains a number of user interfaces. These are displayed in our user manual.

IX. Design Deficiencies

In our implementation, we discovered a number of deficiencies in the design of our project. These are outlined as such:

- some menus and options cannot be clicked while others must be clicked,
- some menus have a slightly different design than others due to limitations of libraries,
- when players create teams in the game setup these are not saved from game to game or when the menu is closed,
- when deleting teams, the teams must be deleted in the reverse order when being put in,
- and there is no way to exit an already started game.

These deficiencies indicates flaws in our design and offer future work on the project.

Additionally, we found a number of deficiencies with missing classes which we needed to create. For example, we needed more views than originally anticipated, and a class to effectively connect our controllers together (a primary controller of sorts).

X. Conclusion

In conclusion, our implementation has made it to the alpha stage of development. After addressing certain design deficiencies and missing features, we will be able to continue to beta development. There were many obstacles that were faced in the development process, which we overcame using techniques such as pair programming, code reviews, consistent code style and a flat group structure. Additionally, our reflections show the kinds of software development realizations we came to during the process.

The work was well divided between all members and there have been no complaints about workload. Although proper testing is not where we want it to be, we hope to have that address as we transition from alpha to beta.

Lastly, looking back on the progress made we are pleased with the work accomplished as well as the functionality we achieved, but have determined a number of features we would like to still see implemented as outlined in the report.