Software Reengineering

Final Report

Ma INF 2020-2021

Igor Schittekat

1 Assignment

The Assignment can be found at
`https://ansymore.uantwerpen.be/2021-reengineering-project`

2 Introduction

For this project, I had to reengineer the project Megamek. A fork of this project which I reengineered can be found on `github.com/IgorSchittekat/megamek/`.
For this project, I worked alone, so every decision that was made to reengineer parts of the code were my own decisions.
In my intermediate report, I decided to take a look at the Entity class and the Server class. When starting the project, this was an obvious choice, because those classes are big god classes and and really needs refactoring. However, I quickly noticed that those classes were way too big for me to figure out on my own. Without a group partner to discuss possible changes, I wouldn't risk taking on such large classes.
I decided to first focus on the class AmmoType, and after that I shifted my focuss to BoardView1. In the end I also took a quick look at DestructionAwareDestinationPathfinder.

3 AmmoType

AmmoType is a class with lots of duplicate code, so I started off this project by reducing some of these duplicates in a way that seemed logical.
Many of the methods of AmmoType are static functions that create specific ammotypes. The problem here is that all those functions are slight modifications of one another. They all set different parameters of the AmmoType class, but some of them vary only in some of those parameters.
I decided to group the ones that are almost the same. For instance the methods to create CLLRM Ammo has 16 different variants, and they only differ in rackSize, bv and kgPerShot (and the names, but those depended on the rackSize, so this could also be fixed). All other parameters were the same. So I created the method createCLLRMAmmo implementing the entire function, added arguments to pass the varying parameters, and called that function from within each of the varying types, passing alont the correct varying parameters.
I decided to only group methods with similar names and different rackSizes together, and no different other methods. I thought about creating one single method setting everything, and calling this method from within each of the different methods. But because there were way too many parameters that had to be set, and not all functions set the same parameters, this would get very messy and didn't contribute to better code. Yes it would drastically reduce the code duplication, but it would also be very hard to read and to manage.
So I opted for an approach of grouping only similar ammo types, only differing by rackSize alongside with a few other parameters. The future advantage for doing it this way is that adding a new similar type of ammo is easy, and doesn't require another copy-paste.

Because I had to manually create those new methods, and check which parts could be moved and which parts had to be replaced with a parameter, many errors could be introduced from my end. The tools provided in the lab were nice to detect the duplicates, but could not resolve them, so I had to rely on my own ability to not make mistakes in checking differences. The tool Dude could help with identifying those differences, but it was not a very nice tool for this file. It was good to point me to this file in the first place, but then it stopped serving his purpose for me, as the duplicates were very obvious. iClone only gave me textual output, as pointed out before in my previous report, so that tool was not any better.
So because there was a very high chance of making mistakes, many tests had to cover the possible errors. I decided to not write my own tests, because it would take way too many time to individually test every single parameter before refactoring. That is were Diffblue Cover came in handy. Diffblue Cover is an IntelliJ plugin I found to automatically generate tests, and it served the exact purpose I needed. It generated test cases for every single one of the static methods, even for the ones that I didn't refactor. This way I could make sure that I didn't introduce errors myself.

Although I refactored many of these methods, I didn't do them all. It was a very time consuming task, and also very repetitive work. After a while I decided to move on to another class, and look for other things than just removing duplicate code.
In the end, I came back to this file to remove more duplicates to optimize the file as much as possible.
I noticed that there are more files like this, which have similar problems. MiscType is one example, where the same refactoring can be done in future work.

4 BoardView1

The BoardView1 class was the other class where I focused my time on. This was a class that couldn't be tested easilly, so I decided to refactor most of the time using the refactor functionality in IntelliJ, ensuring in this way that I didn't introduce side effects in the code.
When analysing the project using CodeScene, I noticed that this file had a code health score of 1/10. CodeScene told me that most of the problems were caused by methods doing too much things. This could be solved by extracting functions and making sure each of them served a single purpose.
To tackle this problem, I used the IntelliJ refactoring method Extract Method, on the parts of the code that looked like they served a single purpose. In some cases I had to shift some code around to make this work, but I always checked if those shifts could be done without causing problems. For instance if a variable was passed to a function, and later used again, I made sure that those orders didn't change, as the function could possibly change the variable, and in that case the outcome could be different.
The next big issue in this file were deep nested structures. Ideally the depth is at most 4, where we look at if statements, loops, etc. But in this file the depth exceeded this limit. I managed to refactor the file in such a way that the depth kept below the limit.
For is-statements this could be solved most of the time by inverting the conditions, and creating a guard for the function instead of putting everything in a big If. For loops or for if-else statements this was more difficult, as these could not be inverted. In this case I looked if it was possible to extract a method. This extracted method would then specifically focus on solving this single task.
To make sure I didn't introduce more errors, I used the IntelliJ build in functionality.

Overall, I managed to increase the code health score to 1.32/10, which was less than expected for the work that I did. This shows that the file is still not at all healthy, and still more refactoring can be done.

5 Smaller changes

While looking through the different files of the project I noticed some smaller things that could be refactored along the way. Most of the time because IntelliJ told that the code could be optimized or there were duplicates in switch statements, and cases could be merged. In some of those cases I decided to refactor those small parts of the code, even though the rest of the file is still messy.
Sometimes I also let IntelliJ reformat the code, if I noticed that the indents were off. This would only contribute

to better readability, which is in my opinion also a big factor. But is is not a major improvement as it could be fixed with a click on a single button.
I also tried to refactor some of the code in DestructionAwareDestinationPathfinder, this because CodeScene pointed out that this file was still very healthy, but the health was decreasing since a few months. So refactoring early would prevent problems in the future. Here I used the same techniques as discussed in the previous section.

6 Future needed improvements

This project needs a lot of future improvements. First of all everything needs to be tested. Because only a very small percentage was tested beforehand, I wasn't able to write very good tests without generating them. If the project was tested enough beforehand, it would be easier to write other tests focusing on the specific reengineering task at hand. I managed to increase the coverage by just 1% for the entire project, but this is still not enough. The initial and final jacoco test report can be found on github under Reengineering Reports.
After analysing CodeScene, the code health of the entire project was initially 1/10. This is really bad. Now the score increased to 1.02/10, which is still really bad. To get the code back to a healthy state, way more files need to be refactored, and possibly code structures need to be changed. Unfortunately this would take a lot of time, and because I didn't have a group partner to discuss such major changes, I decided to stick with one class at a time.

7 Used tools

To start off, I used the tool Dude to check for code duplicates. When I found many in AmmoType, I decided to go and refactor this file.
Next I used CodeScene. This is my most used tool for this project, as it was easy to work with, and it directly points to methods that need to be looked at. The downside was the time it took to analyze the project.
Next I also looked at SonarQube. Like CodeScene, this tool pointed me in the direction of changes that needed to be made. The downside in my opinion was that it is less user friendly. To check the test coverage, I used Jacoco. I run it initially and at the end, so I didn't use it often. I included the reports as a visualization of the test coverage.
To generate some of the tests I used Diffblue Cover. This IntelliJ plugin has a free trail that I could use, and it came in handy for generating the tests for AmmoType.
Finally, the most used tool of all, was IntelliJ. The build in functionality in the IDE to refactor code worked perfect for my use. This ensured that I didn't introduce errors, and saved a lot of time as well.

8 Decisions Made

For this project I made some decisions for what to do and what not. First of all I decided not to change the structure of the project, but stick to reengineering classes themselves. At some point after extracting some methods in BoardView1 I looked if it was possible to move those extracted methods to a different class, as they all belonged together, but all those methods needed different data members, so it would not be feasible to extract a class out of it.
Because I didn't move methods between classes, I made the decision not to make UML diagrams. These diagrams would not be of help for restructuring functions in a single class.