Software Reengineering

Final Report Rev 2

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/`.
As this is a report for the retake of this project, I decided to improve upon the code changes I already made during my first attempt, although those changes don't contribute to the solution for the assignment. The report of my first attempt is included in Appendix A for quick reference.

3 Scope description

For the project, we needed to reengineer Megamek in a way that a scoring system for Users and Bots could be added to track scores over different games. Megamek already has an option to run a dedicated server for different games, so my decision for the scoring system would run inside such a server.
The other possibility I didn't go for, is to run the scoring system on top of the program, and have one global database storing every scores of every servers that runs megamek. But I didn't go for this solution because this would involve adding lots more functionality on top of the reengineering of the program, as many of those needed functionalities are not yet included. Some of which are Adding a unique login so each player can track their scores on different servers and Adding a global service to connect to for tracking the scores.
So instead of going for a global scoring system, the scores are tracked per server, and so it can be implemented within the current project.

3.1 Keeping track of the rating

The rating of Users and Bots need to be tracked. Both regular users and bots will internally be Players. This is nice to keep track of the scores, as this can be done for each of the Players individually.

Initially I was thinking about putting the rating system within the Server class, as the server keeps the information of all players over different games. This has lots of positive effects, as well as some negative side effects which needed to be taken care of. The Server can store the rating for each player based on name or id, as these are the two unique things that identify a Player. But the problem here is that the ids can change over different games, and the names don't have to be unique. If multiple names match, MegaMek corrects them so they are unique. In my initial thought this would not be enough, as players could join and leave over different games, and as unique names are only checked per game and not per server, interference with the ratings can still happen. This can be solved by checking uniqueness within a server instead of per game, but this would give more load on the already large Server file.

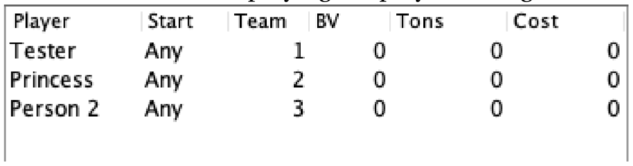
The Server file is a very big god class. And because MegaMek has very bad code coverage I made the decision not to tackle this file. I would have to write many tests for this class before I can even start refactoring this class, and that is way above the scope of this project.
So I scanned through the code to try to find another solution. Initially I thought this couldn't be done, as the rating has to be carried over different games and players must be able to join and leave in between games. The problem here is that Player information is lost when a player leaves, unless a player disconnects and later reconnects due to a network error. But after looking deeper into the Player class, I noticed that players without entities are set to observers, meaning they don't have to leave the game as other players battle.
Using this method, all players and bots can join at the start of the tournament, and keep their ranking over the different games, without the need to store the scores server wide. This has the advantage that the big Server file doesn't need to be touched.

3.2 Updating the ratings

Ratings needs to be updated after each game played. How the ratings are updated is not defined, but winning players get their rating increased and losing players get their rating decreased.

3.3 Displaying the ratings

I decided that the lounge screen when connected to a server would be a good place to display the rating. Currently there is a display with the player's name, their team, the start, the battle value, the tons and the cost. Adding an extra column here displaying the players rating is a nice addition, and keeps everything clean.



Player	Start	Team	BV	Tons	Cost
Tester	Any	1	0	0	0
Princess	Any	2	0	0	0
Person 2	Any	3	0	0	0

4 Design Recovery

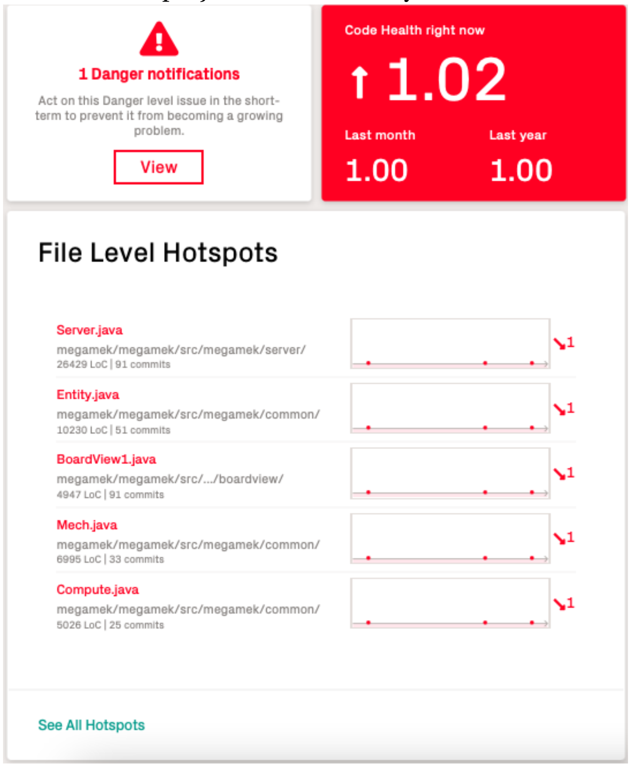
4.1 Important files

For the implementation of the Rating System described above, following files are important:

- Player.java: The Player will have a rating. As both Users and Bots are internally Players, this class can be used to track every rating.
- Game.java: The Game stores all players, and at the end of the game it stores the identity of the victorious team and players.
- ChatLounge.java: This file handles the UI for displaying the rating on the lounge screen.
- BotClient.java: A BotClient controls the bots, and as bots need to get a rating as well, BotCliend will need to be updated to keep bots connected after each game.
- ClientGUI.java: The ClientGUI controlles the gui of a client, and removes bots from the gui at the end of a game. This needs to be changed so bots stay connected.

4.2 CodeScene

As a first look, I revisited CodeScene. Beginning this project CodeScene shows a code health rating of 1.02 for the entire project. This is a very bad score, and I need to take into account that it doesn't get any worse.



1 Danger notifications

Act on this Danger level issue in the short-term to prevent it from becoming a growing problem.

View

Code Health right now

↑ 1.02

Last month 1.00

Last year 1.00

File Level Hotspots

Server.java
megamek/megamek/src/megamek/server/
26429 LoC | 91 commits

Entity.java
megamek/megamek/src/megamek/common/
10230 LoC | 51 commits

BoardView1.java
megamek/megamek/src/.../boardview/
4947 LoC | 91 commits

Mech.java
megamek/megamek/src/megamek/common/
6995 LoC | 33 commits

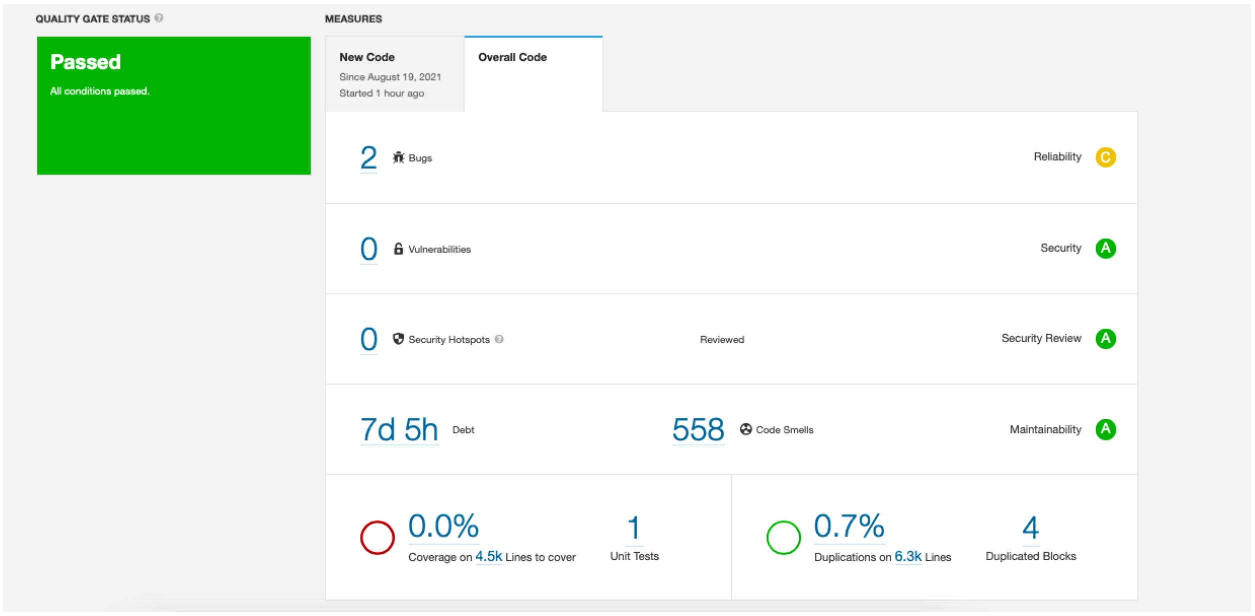
Compute.java
megamek/megamek/src/megamek/common/
5026 LoC | 25 commits

See All Hotspots

Underneath the code health, some really bad files are displayed. One of which is the server file, which I decided not to tackle because of the bad code coverage. I could do major harm to the entire project if I make too many changes here.
The other files displayed here don't influence the rating system, so I don't need to worry about them.

4.3 SonarQube

SonarQube didn't ran well on my system when including the entire project, or even large files like Server.java. So I had to narrow it down to the files that are important for the Rating System. The quality of this code is still very good, but it indicates there are a lot of vulnerabilities. I will take them into account when redesigning.


QUALITY GATE STATUS

Passed
All conditions passed.

MEASURES

New Code
Since August 19, 2021
Started 1 hour ago

Overall Code

2 Bugs — Reliability C

0 Vulnerabilities — Security A

0 Security Hotspots — Reviewed — Security Review A

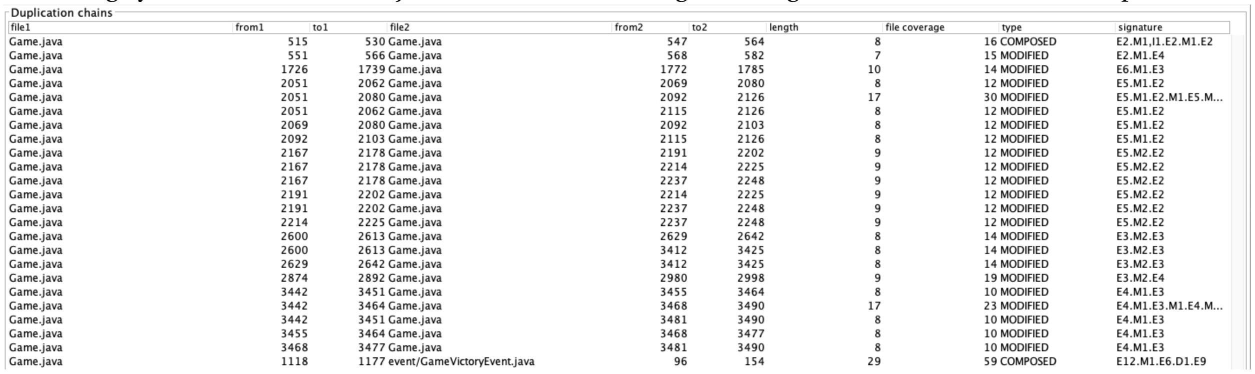
7d 5h Debt — 558 Code Smells — Maintainability A

0.0% Coverage on 4.5k Lines to cover — 1 Unit Tests

0.7% Duplications on 6.3k Lines — 4 Duplicated Blocks

4.4 Dude

With Dude I was able to find duplicates in the code. I found out that there were some duplicates over different files, but I decided that I would only look within the files that I would change.
The Game class was one with lots of duplicates. Although I don't need to make changes to this class to implement the Rating System, I use a Game object to calculate the ratings, so I might as well check out these duplicates.


Duplication chains

file1	from1	to1	file2	from2	to2	length	file coverage	type	signature
Game.java	515	530	Game.java	547	564	8	16	COMPOSED	E2.M1,I1.E2.M1.E2
Game.java	551	566	Game.java	568	582	7	15	MODIFIED	E2.M1.E4
Game.java	1726	1739	Game.java	1772	1785	10	14	MODIFIED	E6.M1.E3
Game.java	2051	2062	Game.java	2069	2080	8	12	MODIFIED	E5.M1.E2
Game.java	2051	2080	Game.java	2092	2126	17	30	MODIFIED	E5.M1.E2.M1.E5.M...
Game.java	2051	2062	Game.java	2115	2126	8	12	MODIFIED	E5.M1.E2
Game.java	2069	2080	Game.java	2092	2103	8	12	MODIFIED	E5.M1.E2
Game.java	2092	2103	Game.java	2115	2126	8	12	MODIFIED	E5.M1.E2
Game.java	2167	2178	Game.java	2191	2202	9	12	MODIFIED	E5.M2.E2
Game.java	2167	2178	Game.java	2214	2225	9	12	MODIFIED	E5.M2.E2
Game.java	2167	2178	Game.java	2237	2248	9	12	MODIFIED	E5.M2.E2
Game.java	2191	2202	Game.java	2214	2225	9	12	MODIFIED	E5.M2.E2
Game.java	2191	2202	Game.java	2237	2248	9	12	MODIFIED	E5.M2.E2
Game.java	2214	2225	Game.java	2237	2248	9	12	MODIFIED	E5.M2.E2
Game.java	2600	2613	Game.java	2629	2642	8	14	MODIFIED	E3.M2.E3
Game.java	2600	2613	Game.java	3412	3425	8	14	MODIFIED	E3.M2.E3
Game.java	2629	2642	Game.java	3412	3425	8	14	MODIFIED	E3.M2.E3
Game.java	2874	2892	Game.java	2980	2998	9	19	MODIFIED	E3.M2.E4
Game.java	3442	3451	Game.java	3455	3464	8	10	MODIFIED	E4.M1.E3
Game.java	3442	3464	Game.java	3468	3490	17	23	MODIFIED	E4.M1.E3.M1.E4.M...
Game.java	3442	3451	Game.java	3481	3490	8	10	MODIFIED	E4.M1.E3
Game.java	3455	3464	Game.java	3468	3477	8	10	MODIFIED	E4.M1.E3
Game.java	3468	3477	Game.java	3481	3490	8	10	MODIFIED	E4.M1.E3
Game.java	1118	1177	event/GameVictoryEvent.java	96	154	29	59	COMPOSED	E12.M1.E6.D1.E9

5 Redesign

5.1 Player.java / IPlayer.java

Each Player will get a rating. This rating needs to be updated after each game, and needs to be displayed on the Lounge Screen. This rating needs to be easily accessible to update and be read, so both getters and setters will be created.
Because the logic for updating the ratings has yet to be determined, I decided that it would be best to not include this in the Player, as it might need the scores of the other players to determine the new score.
Currently, the Player class is for the most part a data class, without much logic build into. This is not a big problem, but the project might benefit if some functionality was moved from other classes into this class. For the Rating System however, this is not needed as I explained above.

5.2 Game.java / IGame.java

During a game, nothing needs to happen to the ratings. Only when a game ends, the Ratings will be updated. When a Game ends, we need the game to extract the players, and know which players are victorious and which of them lost the game.

5.3 ChatLounge.java

This class handles the UI display for the lounge. The information about the players is handled by the inner class PlayerTableModel. This inner class uses the clientgui member variable from the ChatLounge class, but next to that no member is called directly.
This inner class can be extracted, and decoupled from ChatLounge by adding a getter for clientgui.
Looking deeper in the ChatLounge class, I noticed that PlayerTableModel is not the only inner class that can be extracted. Although the other classes do not contribute to the Ranking System, I decided to extract them as well, to improves the code health of ChatLounge.

5.4 BotClient.java

When a game ends, the bots are removed from the game. This needs to be changed in order for the rating system to work for bots. Currently, the bots send a message that they will leave, and the connection is closed. This does not need to happen. The connection should be closed when the server closes, and the message is redundant. Instead the bot needs to send a Done indication so the face can end if all users are done.
This change makes it that the behavior of the game changes, but it is necessary in order to add the Rating System in the way I described above.

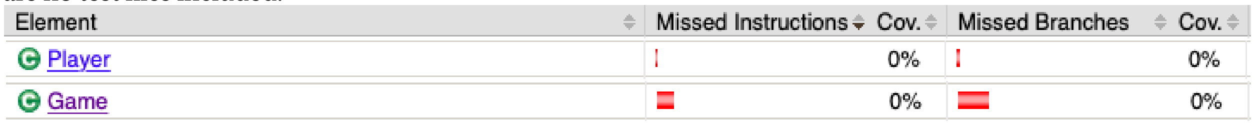
5.5 ClientGUI.java

At the end of the game, the bots are removed from the GUI also. This does not need to happen for the rating system to work. Bots will not be cleared from the GUI.
This change also makes it that the behavior of the game changes, but it is necessary in order to add the Rating System in the way I described above.

6 Management

6.1 Testing

Before refactoring anything, tests needs to be written for the current code. Running jacoco on the initial MegaMek project shows that the coverage of both Player as Game are both 0%, which is expected as for both classes there are no test files included.



Element	Missed Instructions	Cov.	Missed Branches	Cov.
Player	🟥	0%	🟥	0%
Game	🟥	0%	🟥	0%

This is the first thing to tackle. Adding the Rating System to the Player class will not influence the other functionality, but because nothing is tested, I thought it might be good practice to write at least an acceptable number of tests for the class.
For displaying the rating in the ChatLounge, I need to make sure nothing breaks also. I think that adding tests for both Player and Game class will take around 2-4 days.

6.2 Adding Rating System

At first sight, I think that adding the Rating System will be an easy problem to tackle. At the end of the game we pass a Game object to somf function to calculate the new ratings for each Player.

For displaying the Rating System, this can be easily added in the ChatLounge.
The estimated time for this addition is around 1 day.

6.3 Refactoring the used Classes

I will run all tools on the classes I touch when adding the rating system, and refactor them so I don't leave the classes worse than they already are. The time it will take will depend on the class, but I think that overall this changes will take 5-10 days of work.

7 Refactoring

7.1 ChatLounge.java

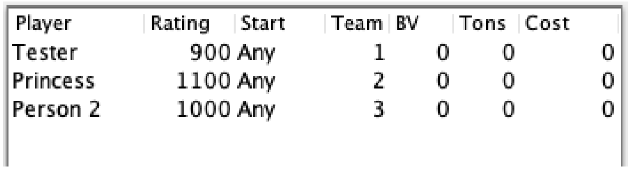
When trying to write tests for Chatlounge and their inner classes before refactoring, I quickly noticed that this was not entirally possible due to the wide use of private variables from ChatLounge within the inner classes. But because ChatLounge manages the UI of the login screen, it was easy to manually test, and check if everything kept working.
I decided to extract the inner classes to their own classes, reducing the load on ChatLounge. Now the Rating System can be added in one of the extracted classes.
All extracted classes are:

- PlayerTableModel (Completed with Rating display)
- PlayerTableMouseAdapter
- MekTableModel
- MekTableKeyAdapter
- MekTableMouseAdapter
- MekTableModelRenderer (Combination form Renderer and MekInfo)

All of these are manually tested before and after the refactoring. For PlayerTableModel I also added automated tests, as the rating system is added here.

7.2 PlayerTableModel.java

The PlayerTableModel is one of the extracted classes from ChatLounge. All ui related things regarding the rating are added here. After extracting the class from ChatLounge, and moving some extra functions from ChatLounge to this class, I made the decision on how to show the ratings. I decided that it was nice to add an extra column next to the Player's name, to display the rating. In each of the relevant functions I added the rating, and called for getters of the Players to retrieve the values. I also tweeked the columns with a bit so the headers are all displayed properly.
For testing purposes I made the column variables public. I could do this and not violate the encapsulation rules, because it were static final variables, so they are read-only.
Finally I changed a large if-else statement within the getValueAt() and setupColumnWidths() functions to reduce complexity, as this was marked as a code smell by SonarQube. The final result looks like this:



Player	Rating	Start	Team	BV	Tons	Cost
Tester	900	Any	1	0	0	0
Princess	1100	Any	2	0	0	0
Person 2	1000	Any	3	0	0	0

7.3 Player.java / IPlayer.java

In the Player class I added a rating variable. Initially the value is set to 1000, but this can be changed depending on the exact formulas that will be chosen. I also created a getter and a setter for the rating.
Next I had to change the isObserver() function. When the game was in the Victory phase, the function would always return false. But for observing players, the rating should not change at the end of the game. As the rating is calculated in the Victory phase, the isObserver() function should still return observers as true when they are, even in that phase.
Using the Find Usage function in IntelliJ I skimmed through the code to see if this change would have any influence, but I didn't see any place where this would break other parts. Next I manually tested this while playing the game, using different configurations of games, playing with or without bots. Nothing seemed influenced by the change as well. The observing players are reset in the Lounge phase later. Mining through the software repository, I found out when this change was made (16 years ago, commit de04e04). Looking at the relevant code at that point in time, I concluded that it was safe to remove the Victory phase check.
Finally I fixed some code smells I found using SonarQube.

7.4 RatingSystem.java

The RatingSystem class is the class that implements the formulas for the rating system. I decided to create a static function that can be called to calculate the ratings for each player in a game. This way, we can check is players are observers, so their rating would not change. The exact formulas should still be added, but for testing purposes I created a simple formula where each victor gets a positive rating of +100, and each loser gets a negative rating of -100, independent of the other player's score.
Additionally I added a private constructor to hide the implicit public constructor.

7.5 Server.java

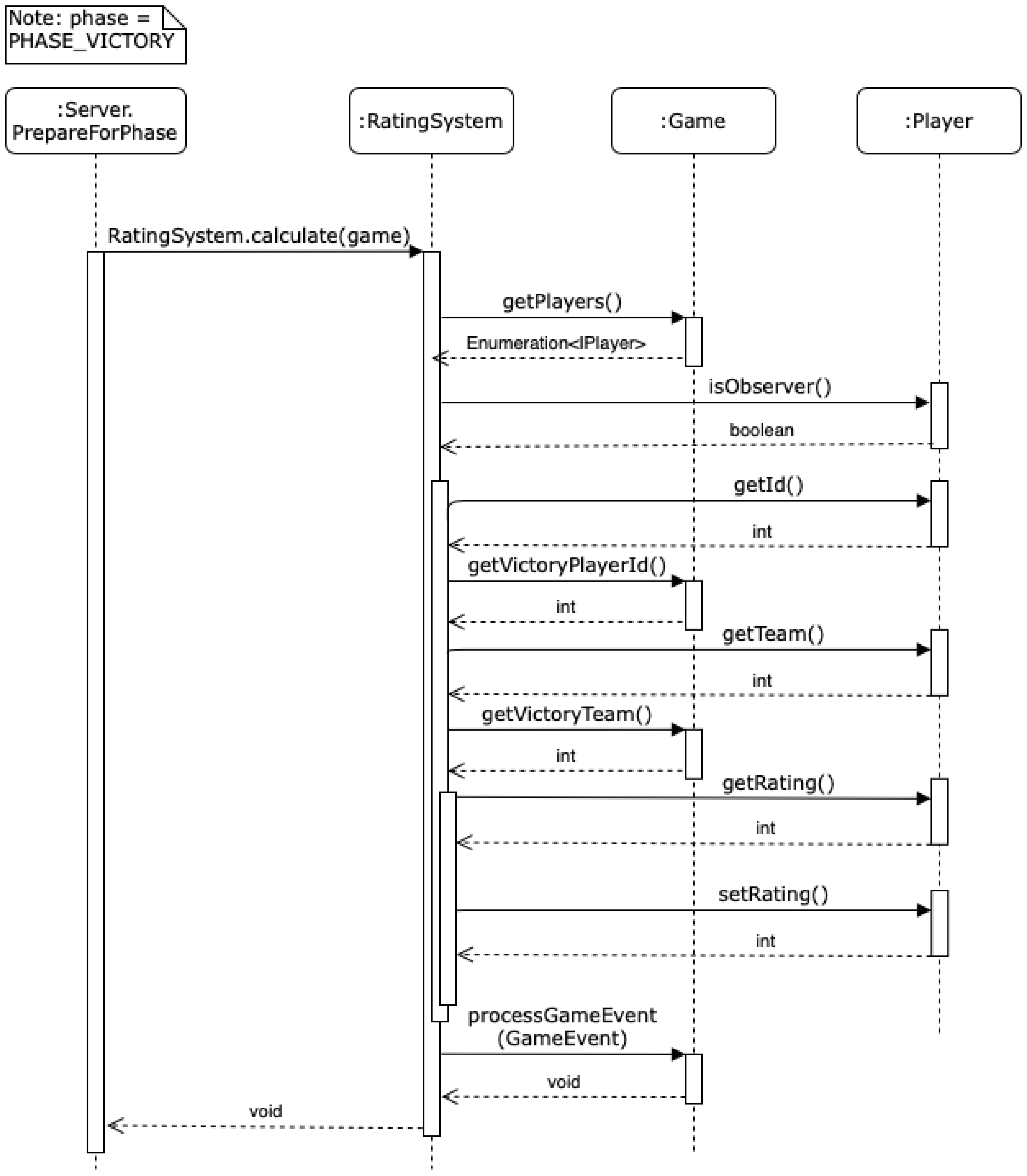
I made the decision not to touch the Server class, because is is very large and was not properly tested beforehand, and to make changes I would first have to add an immense number of tests. But I ran into some issues and came to the conclusion that the easiest fix was to call the rating system update from within the Server class. During the Victory phase I call the RatingSystem.calculate() function with the current game, to update the players within the game.
I manually checked if this addition worked, and it turned out as I hoped it would.

7.6 Game.java

For the rating system to work, I did not have to refactor the Game class. But because the rating system uses a Game object to update their Players, I decided to take a quick look at the functions I used from this class. I removed some unnecessary boxing in those functions, after writing tests for it. I also ran SonarQube on the Game class, and fixed the only critical bug that was pointed out.
After running Dude, I noticed that there was some duplicate code in the Game class. Because I didn't find much duplicate code in the other parts that I used, I decided to remove those duplicates here.

8 The working of the Rating System

When a Player joins, he gets an initial rating of 1000. This is a placeholder value and cal later be changed in the Player class.
At the end of the game, in the Victory Phase, the Server will send an update to the RatingSystem to calculate the new ratings. Currently this is also a placeholder, and can be changed in the RatingSystem class.
The RatingSystem will increase the rating value of each victorious player, and decrease the rating of each loser. Observing Players will keep their rating, as they didn't play this game, and thus their rating should not change.

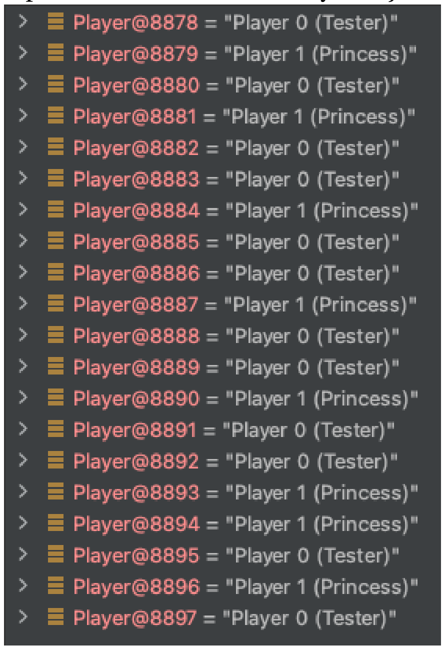


The Ratings will be displayed in the Lounge screen. This is handled by the PlayerTableModel. The PlayerTableModel will call getRating() on each Player, but doesn't make changes to the Rating.

9 Issues with MegaMek

During the project, I noticed some issues with MegaMek. I want to point them out here, because I didn't find a fix, or the fix would take too long to implement and was way out of the scope of the project.

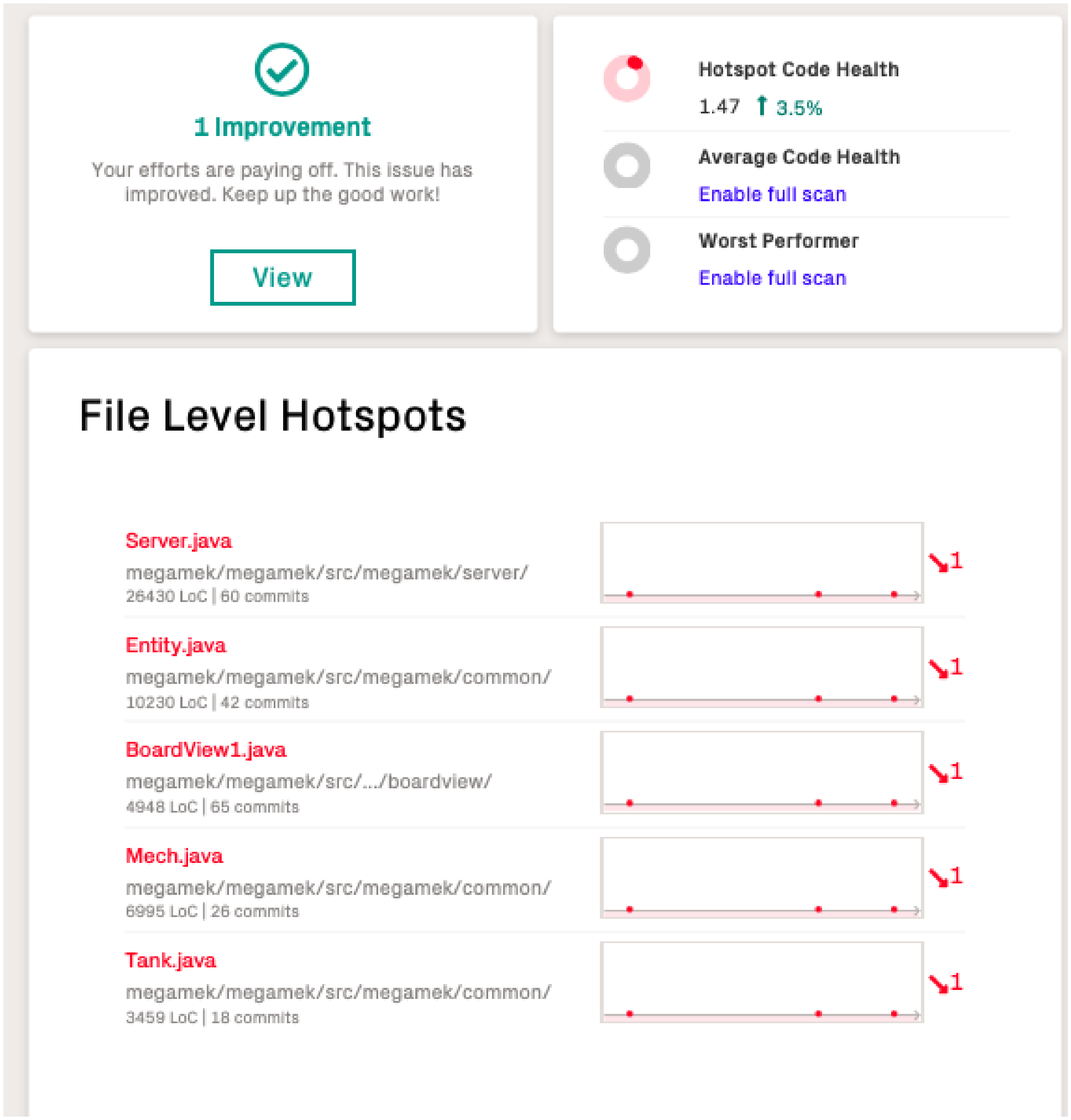
MegaMek is divided into the Server and the Client. When a player joins a server, a Player object is created and added to the Game object within the Server. So for each User or Bot that joins or was created, one new Player object is stored in memory.
MegaMek works with packets to transfer information between Server and Client, and additionally between UI as well, and maybe other parts in the project. These packets can contain objects, such as Players. To detect the exact object, MegaMek uses the library SerialKiller.
SerialKiller receives the packet data as InputStream and creates and ObjectInputStream, and generates a Packet with the correct type of object within the Packet.
The code for this is located in NativeSerializationMarshaller.java, within the function unmarshall().
Now the problem with this is that instead of working with the object that was created before firing the packet, it creates a new object based on the info of the original packet, meaning that a direct copy is made. This makes it that the Server and the Client, as well as the ClientGUI store different Player objects, all copies from each other. Changing the Player in the Server will dynamically update the players in the other location, but not visa versa. This is resolved using Events that try to fix it, but I found out it doesn't always work.
My knowledge on java and events doesn't reach that far that I could resolve why this issue kept occurring. But that is why I made the decision of updating the Rating within the Server, and not within the Client.
The best way to fix this in my opinion is not to work with SerialKiller, and keeping only one Player object in memory for each user or bot, instead of the many objects that are created now. This would probably also reduce memory leaks.
A picture of the different Player objects in memory after one game with a single User and one Boy:



```
Player@8878 = "Player 0 (Tester)"
Player@8879 = "Player 1 (Princess)"
Player@8880 = "Player 0 (Tester)"
Player@8881 = "Player 1 (Princess)"
Player@8882 = "Player 0 (Tester)"
Player@8883 = "Player 0 (Tester)"
Player@8884 = "Player 1 (Princess)"
Player@8885 = "Player 0 (Tester)"
Player@8886 = "Player 0 (Tester)"
Player@8887 = "Player 1 (Princess)"
Player@8888 = "Player 0 (Tester)"
Player@8889 = "Player 0 (Tester)"
Player@8890 = "Player 1 (Princess)"
Player@8891 = "Player 0 (Tester)"
Player@8892 = "Player 0 (Tester)"
Player@8893 = "Player 1 (Princess)"
Player@8894 = "Player 1 (Princess)"
Player@8895 = "Player 0 (Tester)"
Player@8896 = "Player 1 (Princess)"
Player@8897 = "Player 0 (Tester)"
```

10 Final results

To finish of the project, I reevaluated the code health. CodeScene showed that the overall code health improved to 1.47, which is still bad but better than the original score of 1.02. It also showed one bigger improvement, namely in ChatLounge. Here the code improved from 1 to 2.26. This is still a bad score, but this is normal as I only extracted some classes here, and didn't touch the file any further.



1 Improvement

Your efforts are paying off. This issue has improved. Keep up the good work!

View

Hotspot Code Health
1.47 ↑ 3.5%

Average Code Health
Enable full scan

Worst Performer
Enable full scan

File Level Hotspots

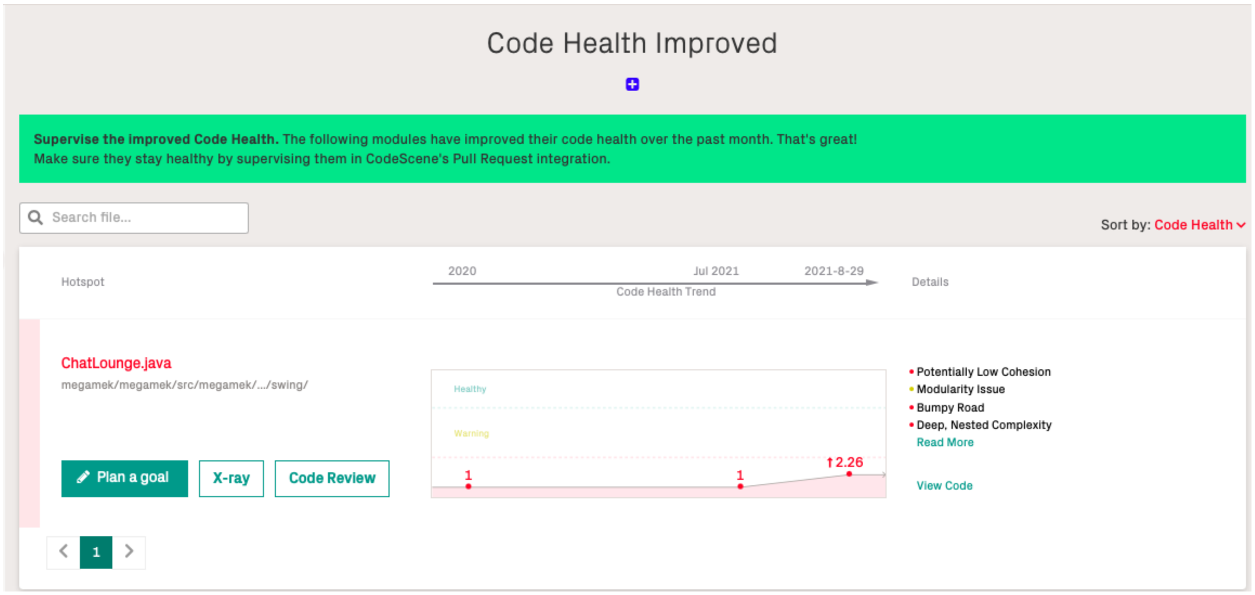
Server.java
megamek/megamek/src/megamek/server/
26430 LoC | 60 commits
↘1

Entity.java
megamek/megamek/src/megamek/common/
10230 LoC | 42 commits
↘1

BoardView1.java
megamek/megamek/src/.../boardview/
4948 LoC | 65 commits
↘1

Mech.java
megamek/megamek/src/megamek/common/
6995 LoC | 26 commits
↘1

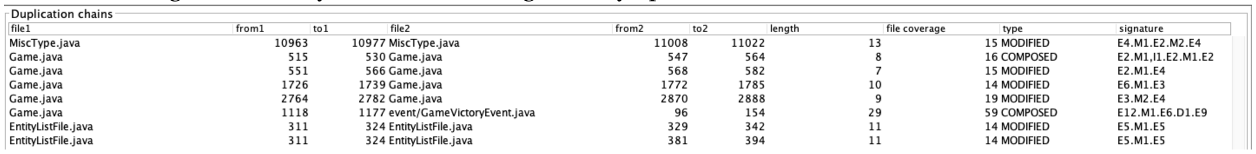
Tank.java
megamek/megamek/src/megamek/common/
3459 LoC | 18 commits
↘1


Code Health Improved

Supervise the Improved Code Health. The following modules have improved their code health over the past month. That's great!
Make sure they stay healthy by supervising them in CodeScene's Pull Request integration.

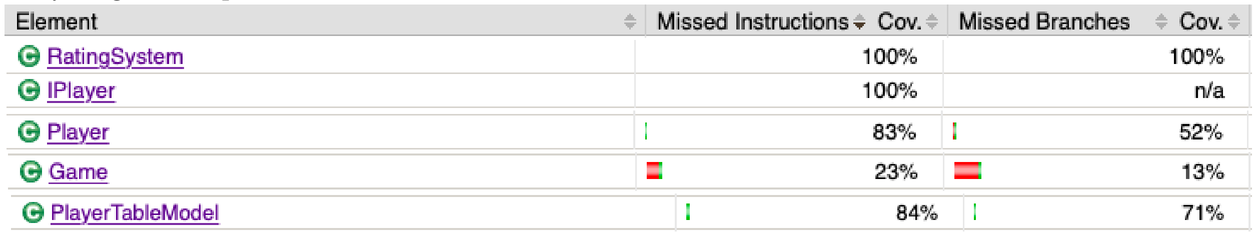
- Potentially Low Cohesion
- Modularity Issue
- Bumpy Road
- Deep, Nested Complexity

Running Dude again showed that most of the duplicates from the Game class are removed. There are still a few left, but looking at them they didn't need changes in my opinion.


Duplication chains

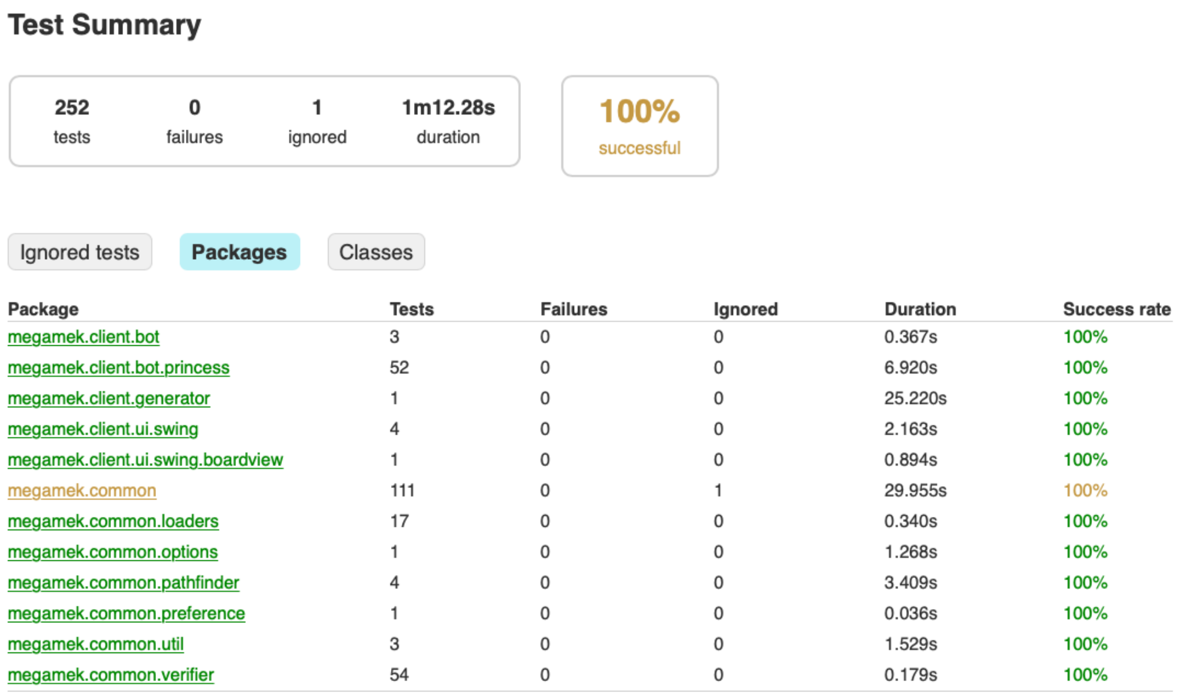
file1	from1	to1	file2	from2	to2	length	file coverage	type	signature
MiscType.java	10963	10977	MiscType.java	11008	11022	13	15	MODIFIED	E4.M1.E2.M2.E4
Game.java	515	530	Game.java	547	564	8	16	COMPOSED	E2.M1,I1.E2.M1.E2
Game.java	551	566	Game.java	568	582	7	15	MODIFIED	E2.M1.E4
Game.java	1726	1739	Game.java	1772	1785	10	14	MODIFIED	E6.M1.E3
Game.java	2764	2782	Game.java	2870	2888	9	19	MODIFIED	E3.M2.E4
Game.java	1118	1177	event/GameVictoryEvent.java	96	154	29	59	COMPOSED	E12.M1.E6.D1.E9
EntityListFile.java	311	324	EntityListFile.java	329	342	11	14	MODIFIED	E5.M1.E5
EntityListFile.java	311	324	EntityListFile.java	381	394	11	14	MODIFIED	E5.M1.E5

Finally I ran Jacoco on all tests. As none of the files that I touched were tested beforehand, this was a big improvement. The test summary and the jacoco test report are included in the github repo, as I couldn't include everything in the report.



Element	Missed Instructions	Cov.	Missed Branches	Cov.
RatingSystem		100%		100%
IPlayer		100%		n/a
Player		83%		52%
Game		23%		13%
PlayerTableModel		84%		71%

Game is only tested for 23%. This is because I only tested the parts that I refactored, and didn't cover much of the other code.


Test Summary

252	0	1	1m12.28s	100%
tests	failures	ignored	duration	successful

Ignored tests | Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
megamek.client.bot	3	0	0	0.367s	100%
megamek.client.bot.princess	52	0	0	6.920s	100%
megamek.client.generator	1	0	0	25.220s	100%
megamek.client.ui.swing	4	0	0	2.163s	100%
megamek.client.ui.swing.boardview	1	0	0	0.894s	100%
megamek.common	111	0	1	29.955s	100%
megamek.common.loaders	17	0	0	0.340s	100%
megamek.common.options	1	0	0	1.268s	100%
megamek.common.pathfinder	4	0	0	3.409s	100%
megamek.common.preference	1	0	0	0.036s	100%
megamek.common.util	3	0	0	1.529s	100%
megamek.common.verifier	54	0	0	0.179s	100%

11 Future work

The MegaMek project has still a lot of issues that need to be resolved. First of all, the testbase needs to be increased, as there are still many places that are poorly tested.
Next, there are still many duplicates found in the codebase. I managed to reduce the number in the Game class, but there are many more classes with even more duplicates.
Regarding the Rating System, the formulas need to be updated and re-tested. Currently a placeholder is in place to show that the rating system works, but this is just a simple version of one.

Appendix A

This was my first report of all changes I made for the project in the first take. This is included for quick reference, but does not contribute to the reengineering task from the assignment. I did not make any changes to this part of the report.

Introduction

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.

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.

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.

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.

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.

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.

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.