Software Reengineering: Report

Max Van Houcke Thomas Van Onsem

May 2021

1 Project

- MegaMek
- Max Van Houcke & Thomas Van Onsem
- GitHub: https://github.com/ThomasVanOnsem/megamek

2 Setting Direction

2.1 Context

MegaMek is a tactical turn-based game where users can play against human opponents or built-in AI. We have been asked to re-engineer the project to enable a swift implementation of a ranking system for both users and bots.

Based on existing ranking systems, we decided to update a player's ranking every time a game finishes. This means we focus on both the game and the player classes (PATTERN: Most Valuable First). Since a player ranking is used across different servers, the storage of that data would also be part of the system. This will also be kept in mind during the re-engineering steps.

2.2 Management

Before we started, we agreed on a couple of things. We hold weekly meetings (PATTERN: Speak to the Round Table). Since we are a team of 2, we'll both act as navigators, correcting each other during development. (PATTERN: Appoint Navigator)

2.3 Risks

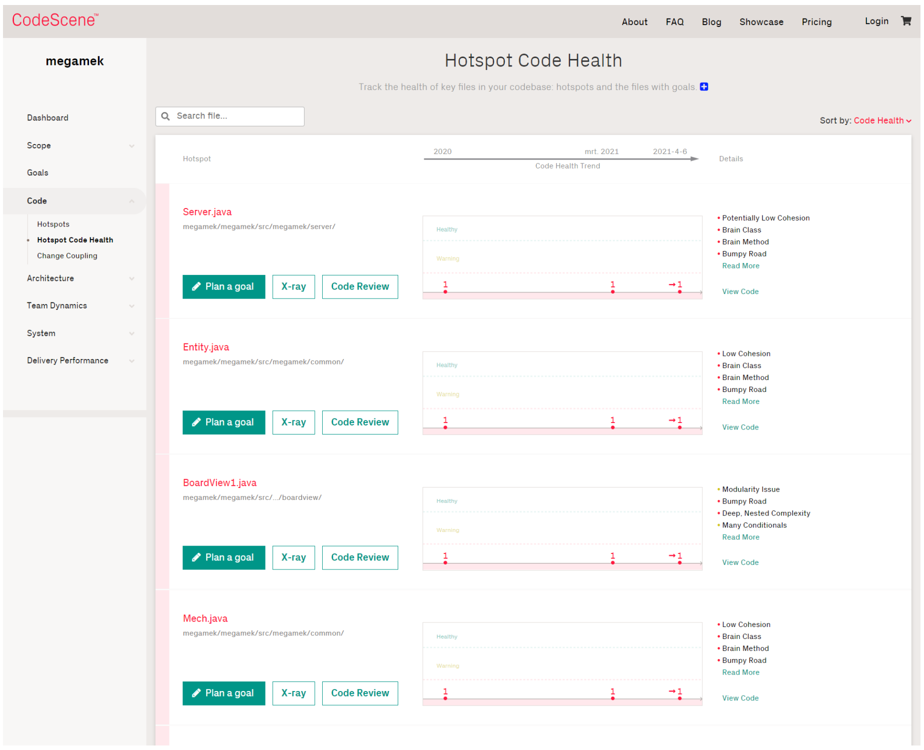
We have to make sure the program still works without a competitive setting. It also needs to be possible to completely change the calculation of the score without the program breaking.

3 Design Recovery

3.1 First Contact

3.1.1 CodeScene

CodeScene reports that our project has a code health score of 1.27 (marked as bad), a rapid code health decline for BoardEditor.java and a couple of problem files (shown in the screenshot below).



Hotspot Code Health

Track the health of key files in your codebase: hotspots and the files with goals.

Server.java
megamek/megamek/src/megamek/server/
- Potentially Low Cohesion
- Brain Class
- Brain Method
- Bumpy Road

Entity.java
megamek/megamek/src/megamek/common/
- Low Cohesion
- Brain Class
- Brain Method
- Bumpy Road

BoardView1.java
megamek/megamek/src/.../boardview/
- Modularity Issue
- Bumpy Road
- Deep, Nested Complexity
- Many Conditionals

Mech.java
megamek/megamek/src/megamek/common/
- Low Cohesion
- Brain Class
- Brain Method
- Bumpy Road

These files we have to look out for when diving into the system and deciding where to implement our changes.
Since we don't have the upgraded plan for CodeScene, the only other general subject we can check are the team dynamics. This would be useful in a situation where communication with the development team is possible. In our case, however, this does not serve a direct purpose. (PATTERN NOT APPLICABLE: Chat with maintainers, Interview during Demo)

Using the metric tools, we now have a proper overview and initial understanding of the different parts of the system. This enables us to make a quick assessment of the project's magnitude and problem areas.

3.2 Initial Understanding

3.2.1 Scope

During our meetings, we skimmed through the source map, based on the structure found in our first contact, to get a feeling of the area we need to work in (PATTERN: Skim documentation in 1 hour). We ended up marking the following files for our re-engineering efforts (PATTERN: If it ain't broke, don't fix it, Most valuable first):

- src/megamek/common/Player.java (and IPlayer): A player will have a competitive score.
- src/megamek/common/Game.java (and IGame): We update our score after the end of a game.
- src/megamek/common/event/GameListener.java: A listener will probably be needed to notice the end of a game.
- src/megamek/client/Client.java: Client and Players are connected, client will have to pass either a ranking or a unique identifier so the ranking can be retrieved.
- src/megamek/client/Server.java: A server can be competitive or standard, this option will be set in the Server class.
- tests/: To be detected when running test coverage tools.

The major Server.java file was analysed (PATTERN: Study exceptional entities) and after some discussion included in our reengineering. Most of the logic can be implemented in the Game class, but the Server still has to pass the competitive option to the Game it manages.

3.3 Detailed Model Capture

Because of the huge amount of code in the project, and the lack of understanding how our key classes (PATTERN: Most valuable first) are being used, we decided to step through the code (PATTERN: Step through the execution) and understand the way the classes are used (PATTERN: Look for the Contracts) resulting in a better general view.

3.3.1 Player

The first class we focus on is Player. We start by stepping through the code. Everything starts at the constructor, so we set our breakpoint there. What we immediately see is the use of a connection to initialize the Player. Even though we're running the game locally, it still sends a packet through a connection to control the Player from the Client. The Player is thus only being used in the game simulation, while the Client sends all the commands using packets.

The class does not contain much logic and acts more as a data class, considering the large amount of getters and setters. When searching for usages with IDE tools, we find usages in the expected Game class but also in UI classes and others.

3.3.2 Game

Next class is Game. Same as previously, we start at the constructor. We see that the class is instantiated in the Server as well as the Client. As documented, these two objects need to be in sync, considering that the class is the root of all game data.

When we look at the usages of Game, as expected we find them in Server, Client and UI classes. We zoom in on the instance in the Server class and find a huge amount of method calls on the object, exceeding a thousand. When looking through a selection of them there are mainly get and set calls. This is in line with our first understanding of Game, being mainly a data class. It also indicates that there's a lot of game logic present in the Server.

3.3.3 Change Frequency

To see how frequently our key classes were changed during development, we perform data mining techniques on the repo. This results in the following data.

File	Touches	LoC
Player	105	589
Game	324	3679
GameListener	5	69
Client	369	1794
Server	2805	35694

It's clear that our key classes have a significant amount of commits associated, especially the Player class considering its lower LoC size. Also noticeable is the Server, which towers above everything with a huge amount of commits.

These results give us more insight into the development of our key classes. We thought about adding more information to this mining process such as authors and dates, but didn't think this would give us any better insights for our re-engineering task.

3.4 System Architecture

Looking at our key classes and the system in general, we recognize some design patterns used throughout.

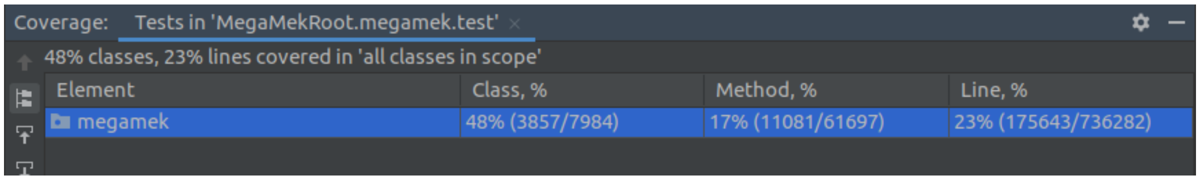
- **Client-Server** As is clear from the classes Client and Server, this pattern is used to run the game on the server and have clients connected to it. Consequentially these classes are a huge part of the project, housing communication (using packets) but also game logic like movement and attack handling.

- **Listener** MegaMek uses multiple listeners such as GameListener, ActionListener and BoardViewListener to notify any subscribed entity of changes. The GameListener is useful in our reengineering, as it can notify the end of a game, the perfect moment to update the players' rankings.
- **Adapter** Another pattern widely used in the project is the adapter, containing default implementations of an interface to make it easy to override only a few. In particular there is a GameListenerAdapter, which as talked about above could be interesting for us.

3.5 Test Analysis

3.5.1 Intellij Coverage

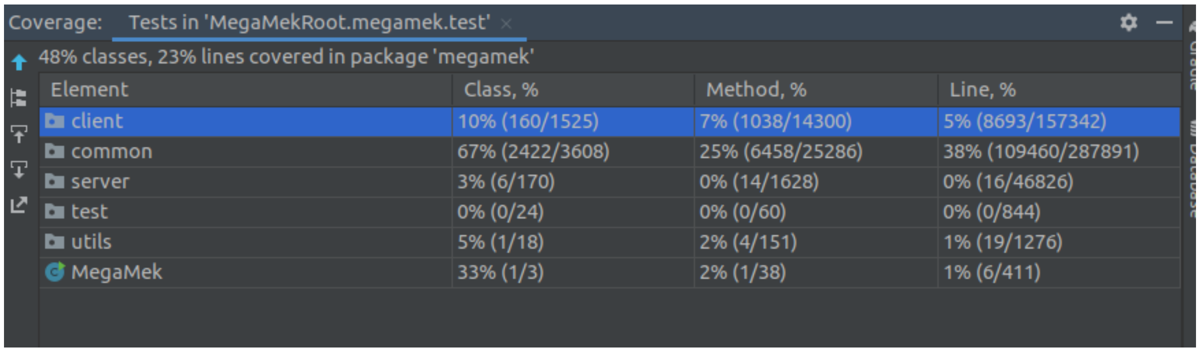
Running the test coverage of Intellij, we can see that the coverage is below average. Only 48% of the general classes are tested. The files within are scope also have low coverage, meaning we should heavily focus on that as well.



Coverage: Tests in 'MegaMekRoot.megamek.test'

48% classes, 23% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
megamek	48% (3857/7984)	17% (11081/61697)	23% (175643/736282)



Coverage: Tests in 'MegaMekRoot.megamek.test'

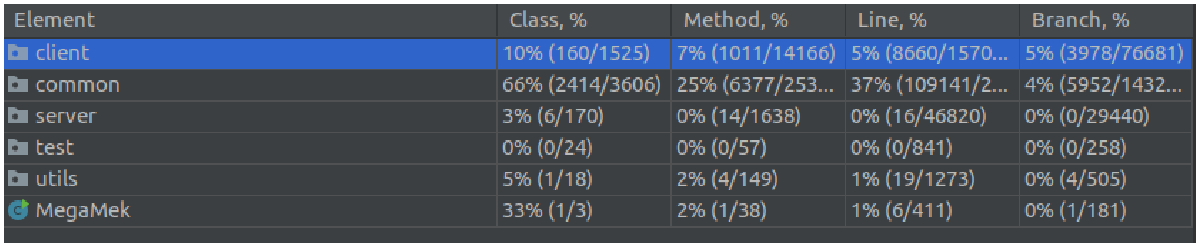
48% classes, 23% lines covered in package 'megamek'

Element	Class, %	Method, %	Line, %
client	10% (160/1525)	7% (1038/14300)	5% (8693/157342)
common	67% (2422/3608)	25% (6458/25286)	38% (109460/287891)
server	3% (6/170)	0% (14/1628)	0% (16/46826)
test	0% (0/24)	0% (0/60)	0% (0/844)
utils	5% (1/18)	2% (4/151)	1% (19/1276)
MegaMek	33% (1/3)	2% (1/38)	1% (6/411)

The Player class has a testing coverage of 0%, Server has 6% coverage, Game has 14% coverage, Client has 33% and the event listeners 14%. This evidently means the tests will not catch any mistakes made in the files we are about to re-engineer.

3.5.2 JaCoCo

Running JaCoCo within Intellij gave us similar results compared to the builtin test coverage. The Player class has a testing coverage of 0%, Server has 6%, Game has 14% coverage, Client has 33% and the event listeners 13%.



Element	Class, %	Method, %	Line, %	Branch, %
client	10% (160/1525)	7% (1011/14166)	5% (8660/1570...	5% (3978/76681)
common	66% (2414/3606)	25% (6377/253...	37% (109141/2...	4% (5952/1432...
server	3% (6/170)	0% (14/1638)	0% (16/46820)	0% (0/29440)
test	0% (0/24)	0% (0/57)	0% (0/841)	0% (0/258)
utils	5% (1/18)	2% (4/149)	1% (19/1273)	0% (4/505)
MegaMek	33% (1/3)	2% (1/38)	1% (6/411)	0% (1/181)

3.5.3 SonarQube

Importing the JaCoCo analysis, gave us the same results as JaCoCo in Intelij. Since the coverage for the files in our scope is very low, SonarQube doesn't have much to offer. However, when increasing the coverage by making some tests, it will produce strong visuals of how good the newly created coverage is. The tool thus needs to be revisited during and after tackling the testing issues.

3.5.4 Little Darwin

Because of the enormous Server.java file, we unfortunately were not able to run Little Darwin.

4 Redesign

4.1 Generic Design

Now that we have a clear view of the system, we can start the redesign process to integrate the new features.

To reiterate, our goal is to add a ranking system. At the moment, all MegaMek players are equal, and there is no notion of experience/level. However, these new features will be optional and a non competitive server should naturally behave as before. Code coverage has to be increased to make sure of this.

Our redesign mainly focuses on the Player and Game classes. The ranking attribute will be added to the Player class, so the Game class can easily access it per player. Because bots are instantiated as Players, they are implicitly included in our design and can have a ranking. Next, we need to update the rankings. A separate RankingCalculator class serves this purpose. At the end of each game, using a GameListener (Design Pattern: Listener), all the game stats are passed to the RankingCalculator which can update the Player's ranking. Because this is a completely different class, it gives the added benefit of being highly flexible with multiple ranking systems (such as ELO).

Not to forget is the initialization of a competitive Game, which will be the task of the Server. To implement this, refactoring will be key considering the class' huge size.

Because rankings should be kept between games and sessions, there has to be some form of persistence. For this we propose 2 solutions: *cloud* or *server* based. Both solutions need a way to uniquely identify the players, using some sort of username. The Client and Server classes would be responsible for this (Design Pattern: Client-Server).

Cloud The cloud solution includes an additional API to store the rankings on a dedicated server. This way, rankings can be securely stored and easily used in any game on any server.

Server Rankings are stored locally on the machine that hosts a server. They can be stored and retrieved anytime a game starts in the same way as the existing game saving options.

The UI part of the ranking system (showing the rankings next to the players etc) and any effects that a ranking may have to the Game are considered out of scope. These are things that should be discussed more with the owners, which can be a logical next step after our re-engineering task. Additionally we expect the most simple of these changes, showing the ranking, to be straight forward once the ranking system is in place.

From this generic design we can start redesigning the test suite and start our refactoring activities. The MegaMek fans are waiting!

4.2 Test Suite Redesign

Concerning the tests, we decided to provide 100% coverage for the methods we create and modify (PATTERN: Write tests to enable evolution). Additionally we try to improve the general coverage for all the important files within our scope.

We do this to make sure we understand all parts we work in (PATTERN: Test to understand) and avoid the introduction of bugs during refactoring and implementation. These tests will be written incrementally with a focus on external behavior to keep tight control over the balance between effort and benefits (PATTERN: Grow the test base incrementally).

4.3 Migration Strategies

To make sure that the eventual changes can smoothly be migrated we follow some principles during the refactoring/development.

- Always Have a Running Version: make sure the main branch can always successfully be built with all tests passing, work on other branches
- Regression Test After Every Change: testing is key, make sure that everything on the main branch always passes

5 Management

5.1 Planning

The following planning has been updated incrementally and gives a main outline of the work we've done and have yet to do. This will be further refined throughout the project.

Phase 1	*Redesign*
March (end)	Player class analysis
April (1st week)	System analysis (tools)
April (2nd week)	Redesign

Phase 2	*Refactoring and Development*
April (end)	Duplicated code analysis
May (1st week)	Finding refactoring targets
May (2nd week)	Prototype and improve scope
May (3rd week)	Refactoring, refine implementation and tests
May (end)	Refactoring, tests and tools

Phase 3	*Rankings*
...	Implement specific ranking system (ELO)

Phase 4	*Persistence*
...	Finalise persistence design
...	Implement persistence

5.2 Effort

We spent about an equal amount of time on system analysis/design recovery and the redesign.

We expect the search of refactoring targets to take some time, as this is the first step to incorporating the ranking system and we are not yet accustomed to actually coding in the MegaMek system. After the end of this phase, our time prediction was indeed correct as this part took the most time.

Next, a considerable amount of time will be spent on tests. However we are more confident that this will go smoothly, as our main goal is increasing the code coverage of Game and Player, both classes we know very well. After the end of this phase, we indeed spent quite some time on tests. However not on the Game class (PATTERN: If it ain't broke, don't fix it).

After our refactoring and test efforts, the implementation of a basic ranking system should be rather straightforward. Thus we do not expect any large problems in *Phase 3*. After the end of *Phase 2*: basic system implemented without much trouble. The actual ELO is for *Phase 3*, but shouldn't be too hard.

Phase 4 planning is not yet finalised, we expect this to be sorted during *Phase 3* of the project.

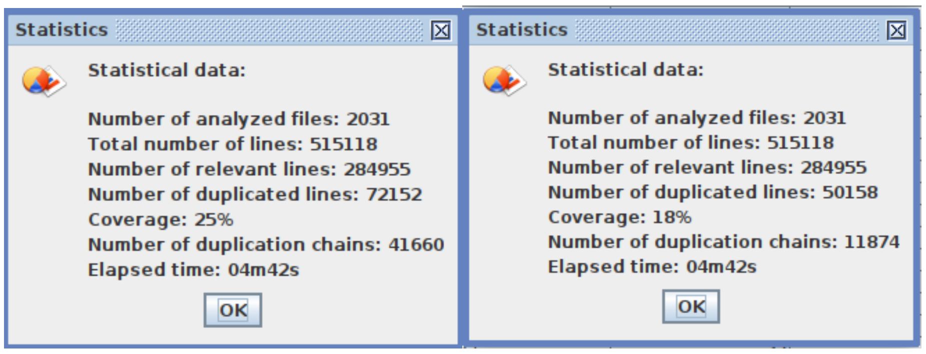
6 Refactoring

6.1 Detecting Duplicated Code

The reports of both analysis tools used in this section can also be consulted on the GitHub repository.

6.1.1 Dude

We know that the industry standard of an acceptable amount of duplication lies between 8 and 12%. Running Dude on the source map with default and more constraining settings, we got the following results (PATTERN: Compare Code Mechanically):



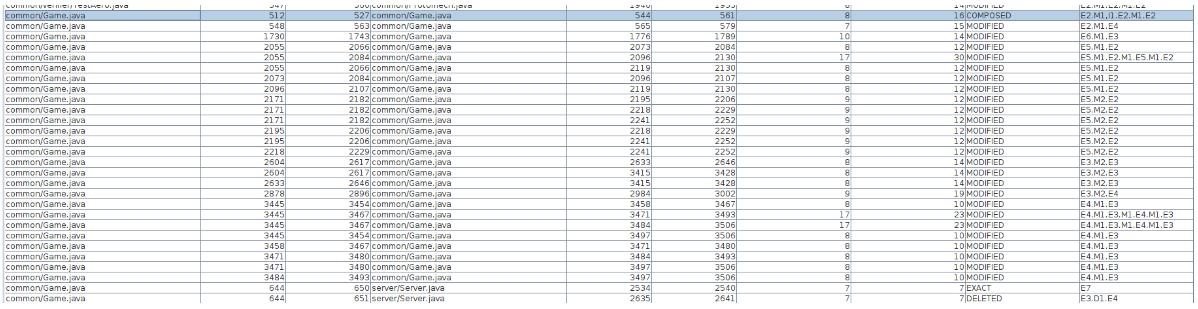
```
Statistical data:

Number of analyzed files: 2031
Total number of lines: 515118
Number of relevant lines: 284955
Number of duplicated lines: 72152
Coverage: 25%
Number of duplication chains: 41660
Elapsed time: 04m42s
```

```
Statistical data:

Number of analyzed files: 2031
Total number of lines: 515118
Number of relevant lines: 284955
Number of duplicated lines: 50158
Coverage: 18%
Number of duplication chains: 11874
Elapsed time: 04m42s
```

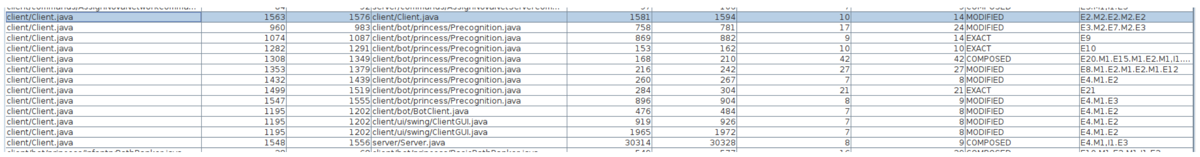
The statistics and some manual inspection indicate some serious code duplication. Looking at the detected files, we notice that most of it was detected between methods of vehicles in the game (e.g.: Between Tank and Dropship) and between weapons. These files do not lie in the scope of our re-engineering task (PATTERN: If it ain't broke, don't fix it).

We do encounter quite a lot of duplicate lines falling within our scope. The server file is the main culprit, having around 500 duplication chains. Client has around 15 duplication entries, Game 20 entries and Player has a duplicate with Team. These are entries we may need to tackle.

After a manual inspection, we concluded that the default settings didn't need to be tweaked that hard, since they only produced a small amount (7 of 100 inspected files) false positive for our files and detected every duplicate we could detect ourselves. Dude analysing the common map with a tokenized distance, didn't get any further than 10%. This is not an issue since we have iClones to analyse the code with a method of similar complexity.



common/Game.java	512	527	common/Game.java	544	561	8	16	COMPOSED	E2.M1,I1.E2.M1.E2
common/Game.java	548	563	common/Game.java	565	579	7	15	MODIFIED	E2.M1.E4
common/Game.java	1730	1743	common/Game.java	1776	1789	10	14	MODIFIED	E6.M1.E3
common/Game.java	2055	2066	common/Game.java	2073	2084	8	12	MODIFIED	E5.M1.E2
common/Game.java	2055	2084	common/Game.java	2096	2130	17	30	MODIFIED	E5.M1.E2.M1.E5.M1.E2
common/Game.java	2055	2066	common/Game.java	2119	2130	8	12	MODIFIED	E5.M1.E2
common/Game.java	2073	2084	common/Game.java	2096	2107	8	12	MODIFIED	E5.M1.E2
common/Game.java	2096	2107	common/Game.java	2119	2130	8	12	MODIFIED	E5.M1.E2
common/Game.java	2171	2182	common/Game.java	2195	2206	9	12	MODIFIED	E5.M2.E2
common/Game.java	2171	2182	common/Game.java	2218	2229	9	12	MODIFIED	E5.M2.E2
common/Game.java	2171	2182	common/Game.java	2241	2252	9	12	MODIFIED	E5.M2.E2
common/Game.java	2195	2206	common/Game.java	2218	2229	9	12	MODIFIED	E5.M2.E2
common/Game.java	2195	2206	common/Game.java	2241	2252	9	12	MODIFIED	E5.M2.E2
common/Game.java	2218	2229	common/Game.java	2241	2252	9	12	MODIFIED	E5.M2.E2
common/Game.java	2604	2617	common/Game.java	2633	2646	8	14	MODIFIED	E3.M2.E3
common/Game.java	2604	2617	common/Game.java	3415	3428	8	14	MODIFIED	E3.M2.E3
common/Game.java	2633	2646	common/Game.java	3415	3428	8	14	MODIFIED	E3.M2.E3
common/Game.java	2878	2896	common/Game.java	2984	3002	9	19	MODIFIED	E3.M2.E4
common/Game.java	3445	3454	common/Game.java	3458	3467	8	10	MODIFIED	E4.M1.E3
common/Game.java	3445	3467	common/Game.java	3471	3493	17	23	MODIFIED	E4.M1.E3.M1.E4.M1.E3
common/Game.java	3445	3467	common/Game.java	3484	3506	17	23	MODIFIED	E4.M1.E3.M1.E4.M1.E3
common/Game.java	3445	3454	common/Game.java	3497	3506	8	10	MODIFIED	E4.M1.E3
common/Game.java	3458	3467	common/Game.java	3471	3480	8	10	MODIFIED	E4.M1.E3
common/Game.java	3471	3480	common/Game.java	3484	3493	8	10	MODIFIED	E4.M1.E3
common/Game.java	3471	3480	common/Game.java	3497	3506	8	10	MODIFIED	E4.M1.E3
common/Game.java	3484	3493	common/Game.java	3497	3506	8	10	MODIFIED	E4.M1.E3
common/Game.java	644	650	server/Server.java	2534	2540	7	7	EXACT	E7
common/Game.java	644	651	server/Server.java	2635	2641	7	7	DELETED	E3.D1.E4

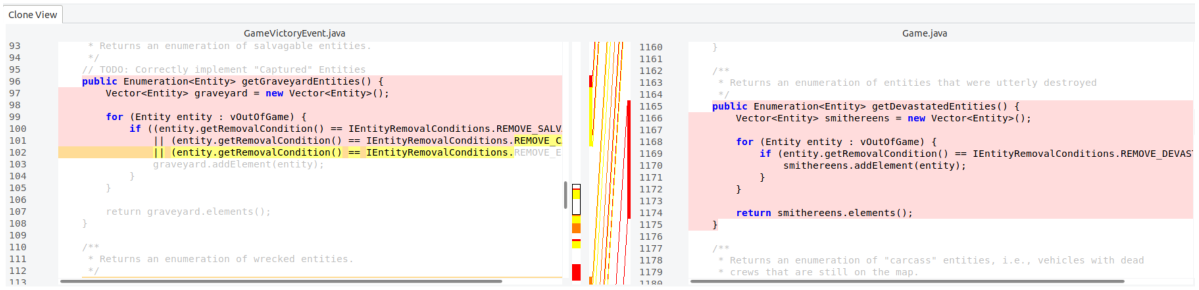


client/Client.java	1563	1576	client/Client.java	1581	1594	10	14	MODIFIED	E2.M2.E2.M2.E2
client/Client.java	960	983	client/bot/princess/Precognition.java	758	781	17	24	MODIFIED	E3.M2.E7.M2.E3
client/Client.java	1074	1087	client/bot/princess/Precognition.java	869	882	9	14	EXACT	E9
client/Client.java	1282	1291	client/bot/princess/Precognition.java	153	162	10	10	EXACT	E10
client/Client.java	1308	1349	client/bot/princess/Precognition.java	168	210	42	42	COMPOSED	E20.M1.E15.M1.E2.M1,I1....
client/Client.java	1353	1379	client/bot/princess/Precognition.java	216	242	27	27	MODIFIED	E8.M1.E2.M1.E2.M1.E12
client/Client.java	1432	1439	client/bot/princess/Precognition.java	260	267	7	8	MODIFIED	E4.M1.E2
client/Client.java	1499	1519	client/bot/princess/Precognition.java	284	304	21	21	EXACT	E21
client/Client.java	1547	1555	client/bot/princess/Precognition.java	896	904	8	9	MODIFIED	E4.M1.E3
client/Client.java	1195	1202	client/bot/BotClient.java	476	484	7	8	MODIFIED	E4.M1.E2
client/Client.java	1195	1202	client/ui/swing/ClientGUI.java	919	926	7	8	MODIFIED	E4.M1.E2
client/Client.java	1195	1202	client/ui/swing/ClientGUI.java	1965	1972	7	8	MODIFIED	E4.M1.E2
client/Client.java	1548	1556	server/Server.java	30314	30328	8	9	COMPOSED	E4.M1,I1.E3

6.1.2 iClones

Similar to Dude, we ran iClones on the source map to get the code quality with respect to code duplication. By default iClones gave 3145 duplication classes mainly consisting of vehicles, entities and weapons. These classes are comparable to the Dude analysis.

Optimizing iClones (with a lower minblock and minclone) gave us some clones not found by Dude, the most important one being a large duplication between Game and the GameVictoryEvent class.

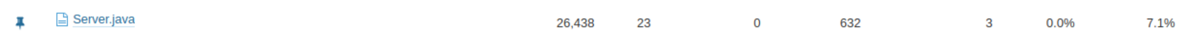


```
     * Returns an enumeration of salvagable entities.
     */
    // TODO: Correctly implement "Captured" Entities
    public Enumeration<Entity> getGraveyardEntities() {
        Vector<Entity> graveyard = new Vector<Entity>();

        for (Entity entity : vOutOfGame) {
            if ((entity.getRemovalCondition() == IEntityRemovalConditions.REMOVE_SALV
                    || (entity.getRemovalCondition() == IEntityRemovalConditions.REMOVE_C
                    || (entity.getRemovalCondition() == IEntityRemovalConditions.REMOVE_E
                    graveyard.addElement(entity);
            }
        }

        return graveyard.elements();
    }

    /**
     * Returns an enumeration of wrecked entities.
     */
```

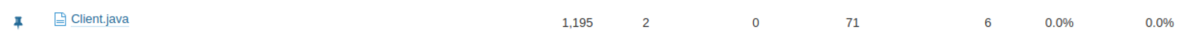
```
    }

    /**
     * Returns an enumeration of entities that were utterly destroyed
     */
    public Enumeration<Entity> getDevastatedEntities() {
        Vector<Entity> smithereens = new Vector<Entity>();

        for (Entity entity : vOutOfGame) {
            if (entity.getRemovalCondition() == IEntityRemovalConditions.REMOVE_DEVAS
                smithereens.addElement(entity);
            }
        }

        return smithereens.elements();
    }

    /**
     * Returns an enumeration of "carcass" entities, i.e., vehicles with dead
     * crews that are still on the map.
```

6.1.3 SonarQube

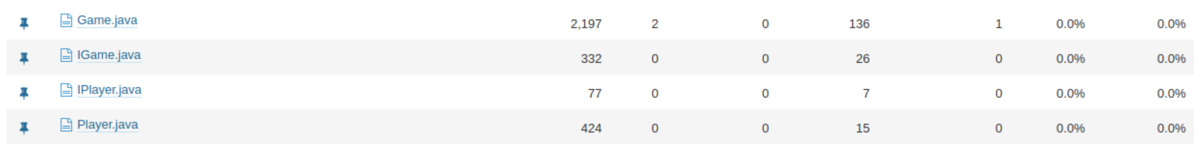
SonarQube is used as a refactoring assistant, indicating code smells, bugs, security issues, duplication and test coverage. However, we ran into a couple of memory issues running SonarQube for the whole MegaMek project. Even when increasing the java heap space to 3GB for the Computer Engine, the Elasticsearch (as recommended by the SonarQube documentation) and the gradle build each, we still got a heap or meta space error. The only way to execute the analysis, was working with file inclusions. This means the statistics we show from SonarQube will only encompass the scope we determined instead of the whole project.



Server.java	26,438	23	0	632	3	0.0%	7.1%

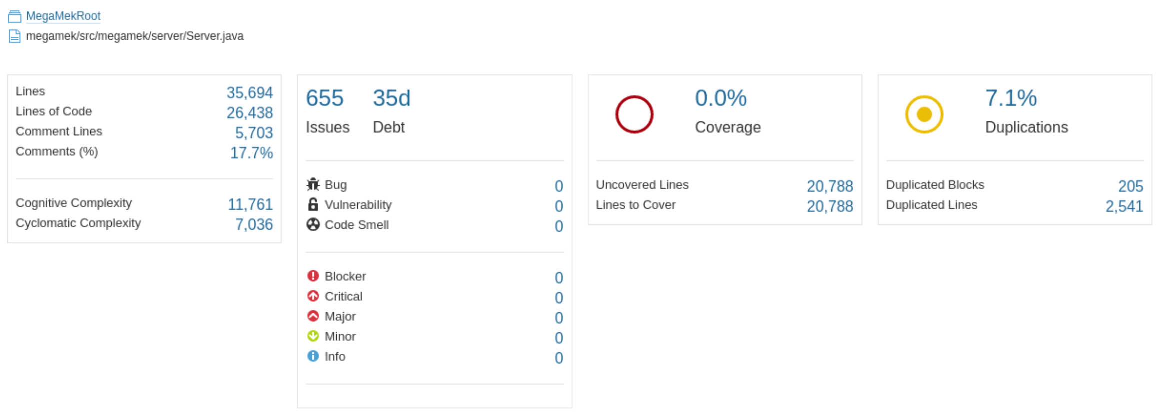


Client.java	1,195	2	0	71	6	0.0%	0.0%



Game.java	2,197	2	0	136	1	0.0%	0.0%
IGame.java	332	0	0	26	0	0.0%	0.0%
IPlayer.java	77	0	0	7	0	0.0%	0.0%
Player.java	424	0	0	15	0	0.0%	0.0%

The results do not surprise us. The other tools also identified the Server class as having the most problems. With most of the critical code smells indicating a high cognitive complexity or a large amount of string literal duplicates.



MegaMekRoot
megamek/src/megamek/server/Server.java

Lines	35,694
Lines of Code	26,438
Comment Lines	5,703
Comments (%)	17.7%
Cognitive Complexity	11,761
Cyclomatic Complexity	7,036

655 Issues 35d Debt

Bug	0
Vulnerability	0
Code Smell	0
Blocker	0
Critical	0
Major	0
Minor	0
Info	0

0.0% Coverage

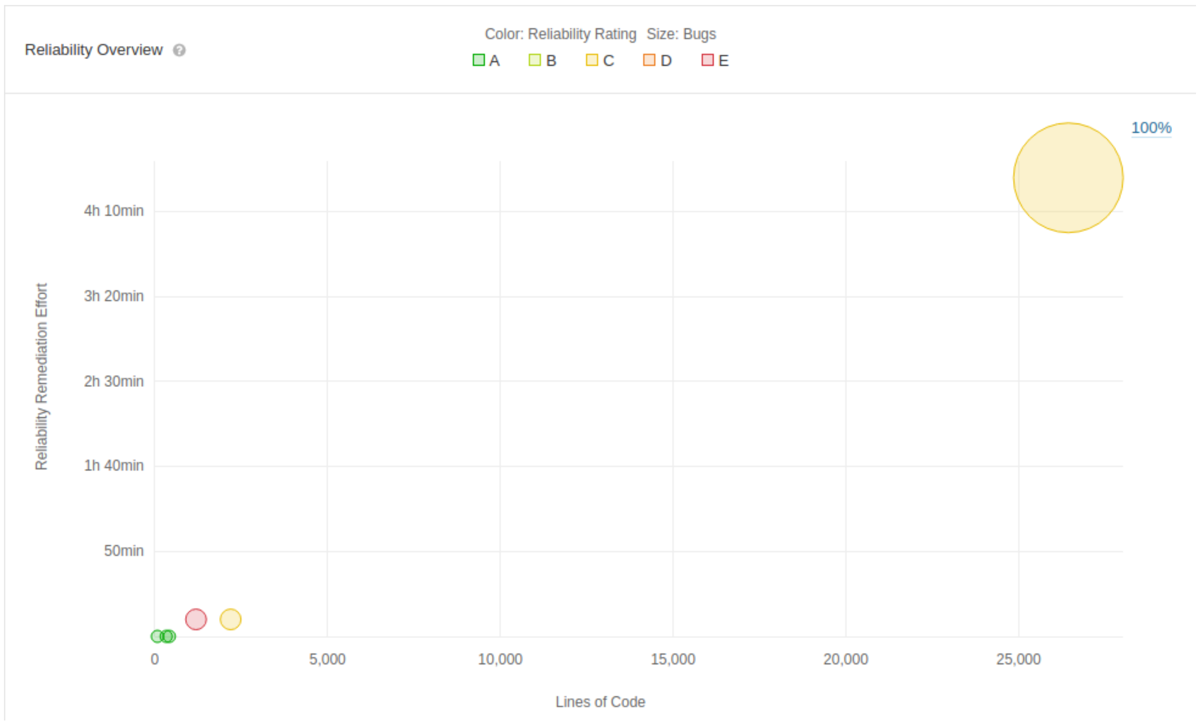
Uncovered Lines	20,788
Lines to Cover	20,788

7.1% Duplications

Duplicated Blocks	205
Duplicated Lines	2,541

Even though the files have a large amount of critical and major code smells, the code (the files within our scope) has an A rating for maintainability with a technical debt of 40 days (2,1%).

The reliability however, seems to be a big issue, producing an E rating mainly because of the Game class (marked in red).



We can conclude that SonarQube gives an excellent overview of all different aspects of the re-engineering process, not only showing the issues but also explaining why those lines are marked as issues, estimating the repair time and visualizing the whole picture. We now know we have to focus our refactoring efforts to the Game, Client and Server class. To deliver a qualitative result, this tool will constantly be revisited during implementation of the changes.

7 Implementation

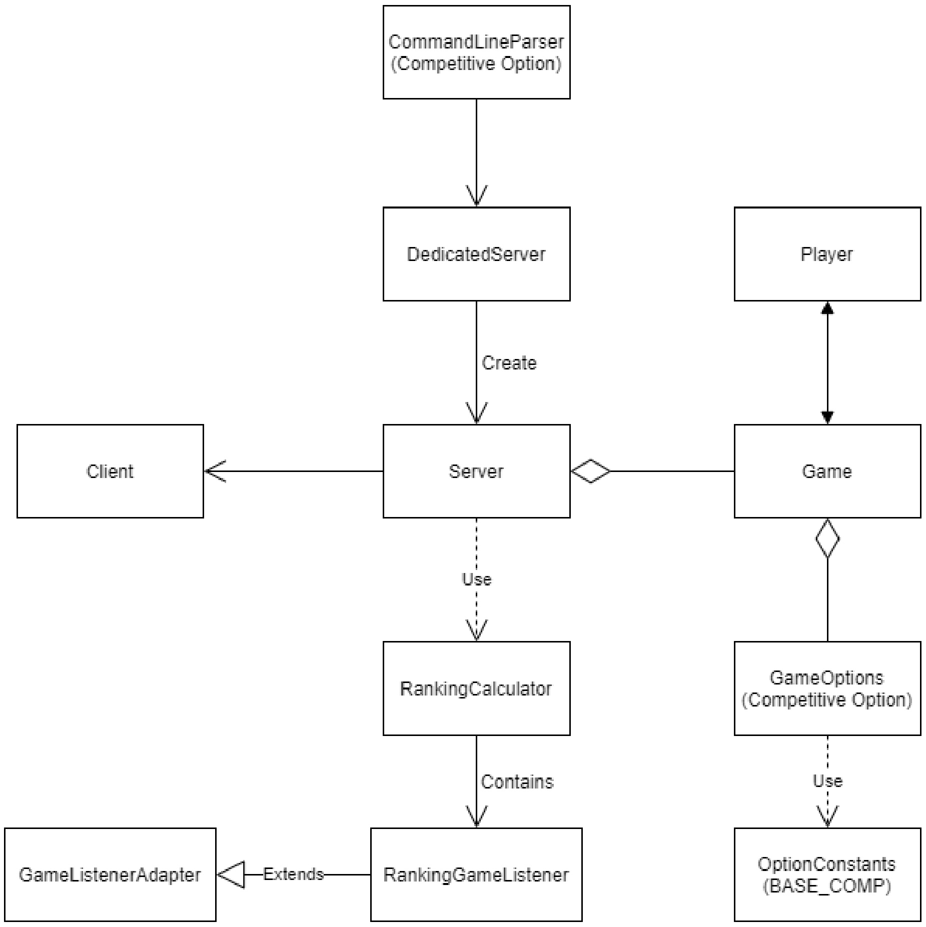
7.1 Adjustment of Scope

After analysis and discussion, the exact scope of refactoring and implementation were still not pinpointed due to the enormous size of the classes in scope such as the Server.

This is why, with our now more in depth knowledge of the system, we both agreed to each make a test implementation of the given assignment (PATTERN: Prototype the target solution). Afterwards, we combined the suggested implementations and adjusted the classes within our scope based on those implementation changes. This meant the Game class became less important. We implemented our ranking system with the use of the following files/classes:

- DedicatedServer: We start a dedicated competitive server with a parameter from the command line.
- Server: A server is competitive or not. It contains 1 game and notifies the ranking system if a user disconnects. It also fetches the competitive score for all players registering with this server.
- GameListener: To avoid having to call the ranking score update in every function containing a game end phase, we opted to use a game listener to get an update on the game ending.
- Player: A player has a competitive score.
- GameOptions: A competitive game option is added such that we don't need to modify the game class itself.
- Client: Contains a password that gets sent to the server for connection with the score database.

This is visualized in the following simplified UML diagram:



These implementations were all done on a separate branch (PATTERN: Always have a running version) and were thoroughly tested. This includes tests for all the existing functions that were touched on our implementation, as to make sure MegaMek still works as normal. After tests were added, the changes were merged to the main branch.

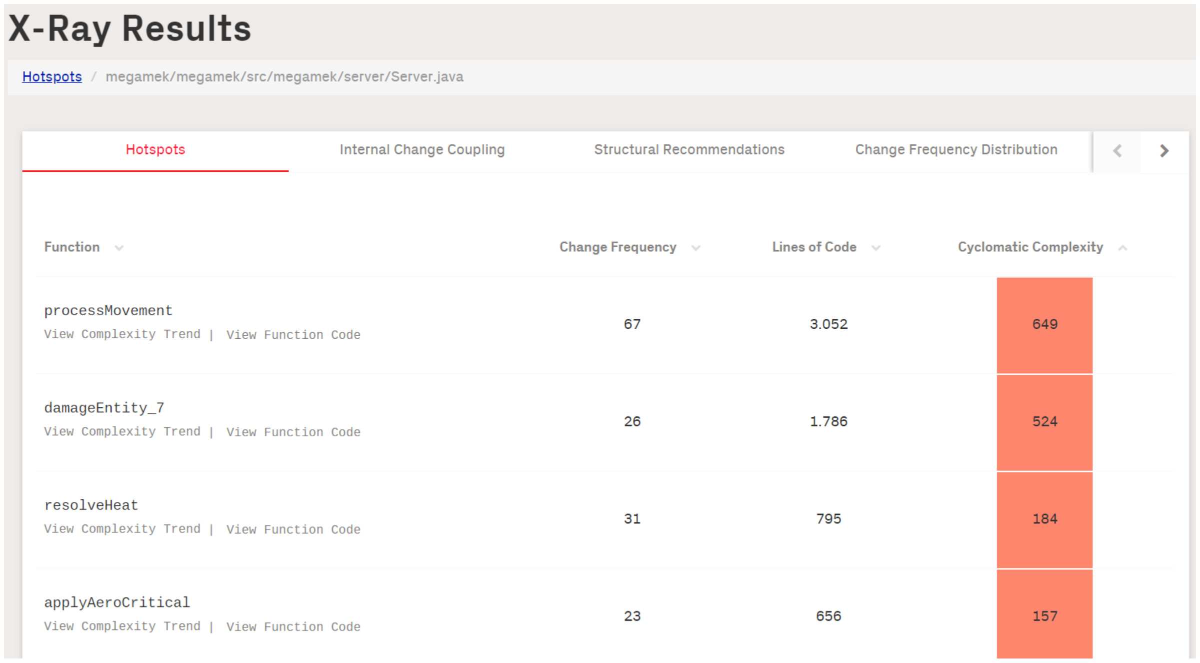
7.2 Considerations

Several tools during our initial analysis indicated the Server file was the biggest culprit on all domains. This meant the file was the topic of most debates during our re-engineering activities. We analysed the whole file and noticed it can be split into 4 major parts:

- Server setup, packet handling, game phase changes (+-5000 lines)
- Movement handling (+-10000 lines)

- Attacks handling (+-8000 lines)
- Damage handling (+-9000 lines)

This is also validated in an X-Ray of the file by CodeScene, where movement, attack and damage functions contain huge amounts of lines, duplication (proven by the analysis done before) and complexity:



X-Ray Results

Hotspots / megamek/megamek/src/megamek/server/Server.java

Function	Change Frequency	Lines of Code	Cyclomatic Complexity
processMovement View Complexity Trend | View Function Code	67	3.052	649
damageEntity_7 View Complexity Trend | View Function Code	26	1.786	524
resolveHeat View Complexity Trend | View Function Code	31	795	184
applyAeroCritical View Complexity Trend | View Function Code	23	656	157

It is clear that these functions, spreading over +- 27000 lines, can all be extracted from the Server class into their own classes (PATTERNS: Redistribute Responsibilities), resulting in a major increase in code quality and decrease in complexity of the Server file. However, these functions have no influence on our ranking system, meaning they do not fall into our scope. And if they did, they would all need to be tested before moving them to avoid breaking the system (PATTERN: Always have a running version). This is why we decided to limit the scope of our refactoring and testing to the functions we added or modified (PATTERN: If it ain't broke, Don't fix it).

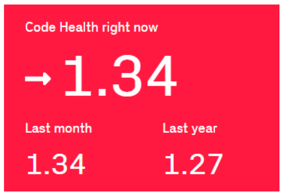
When looking at these functions in our adjusted scope, we quickly realize there are limited refactoring possibilities. Next to these little refactorings, we focus mainly on testing.

8 Evaluation

Now that we have implemented our re-engineering changes, we shall reevaluate the code with the relevant tools used before.

8.1 CodeScene

With CodeScene we can reevaluate the improvements with respect to the whole project instead of focusing on individual parts. Since we only re-engineered functions we touched, we don't expect a major improvement here.



The project still has a bad code health, with a ever so slight increase since we started: +0.07. As mentioned before, the culprit here still is the Server.java file, for which we know how to gravely refactor but mostly didn't touch since it does not lie within our scope.

8.2 Test Coverage

Our main refactoring activity was testing (PATTERN: Write tests to enable evolution), meaning we will see the most improvement in this section. We focused our attention to the complete Player class and all the functions we modified during our implementation (PATTERN: Most valuable first). This resulted in the following increases:

	Before			After		
	Class	Method	Line	Class	Method	Line
Complete Project	48%	17%	23%	50%	20%	24%
Server	6%	0%	0%	33%	8%	1%
Player	0%	0%	0%	100%	100%	100%
Game	14%	1%	3%	42%	20%	18%
GameListener	14%	0%	5%	100%	10%	15%
Client	33%	4%	5%	33%	10%	8%
RankingCalculator	-	-	-	100%	33%	40%

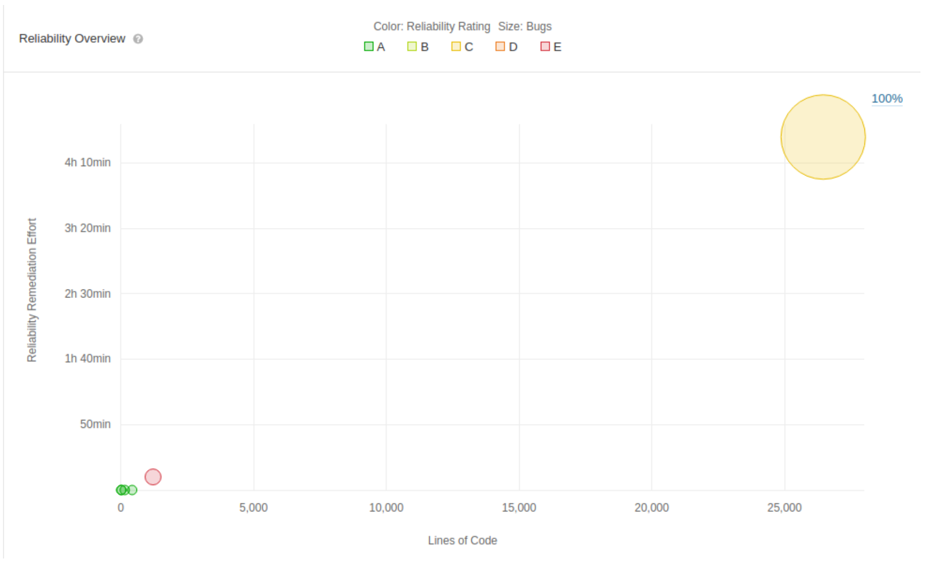
Concerning the RankingCalculator class, it should be noted that we did not reach complete method and line coverage because the functions are left empty. And thus cannot be tested.

It is clear there is a general improvement, certainly for the touched classes.

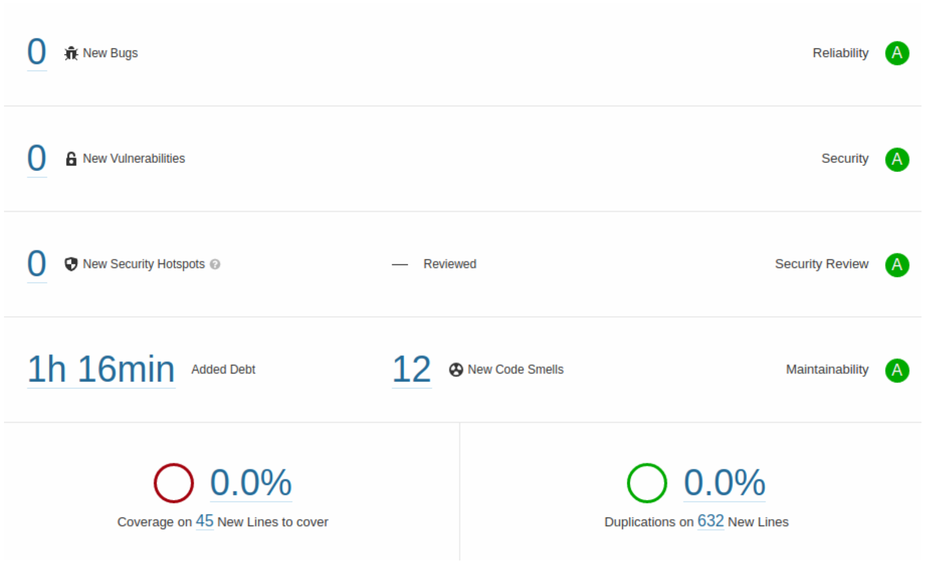
However, the coverage is still poor and should be increased when the scope encompasses the whole project.

8.3 SonarQube

As we discussed earlier, SonarQube gives an almost complete overview of all the re-engineering aspects. But, because of our chosen activities, the only ”big” improvement we will see is with the coverage.



Where SonarQube became useful in our scenario was during analysis of our modifications and implementations. The tool looks at the newly added/changed code and determines how much technical debt we decreased or increased, and collect other statistics on it. With our basic implementation, we noticed we introduced quite a bit of technical debt and 0% coverage because of some code smells together with the not yet imported tests.


0 New Bugs — Reliability A

0 New Vulnerabilities — Security A

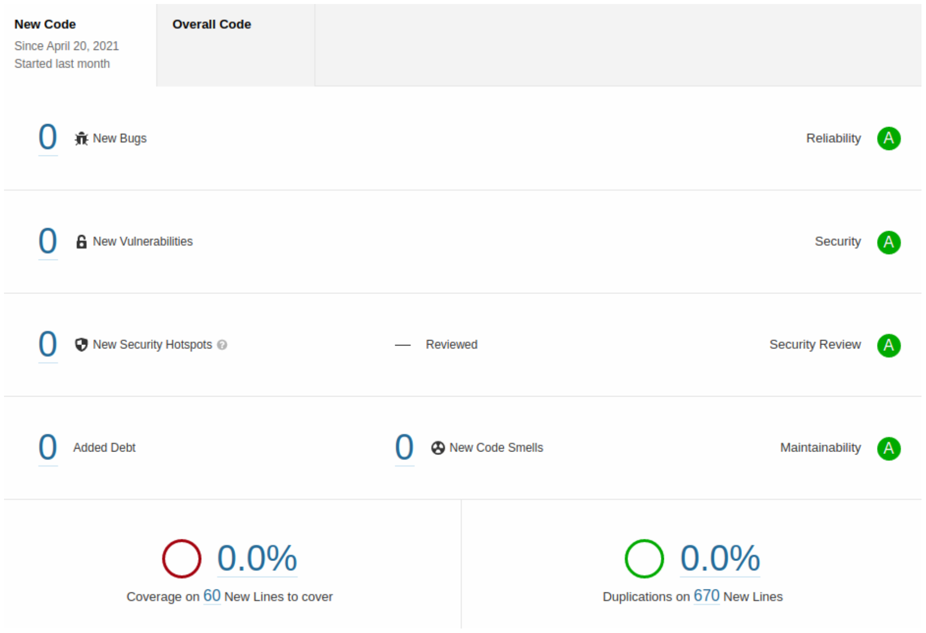
0 New Security Hotspots — Reviewed — Security Review A

1h 16min Added Debt — 12 New Code Smells — Maintainability A

0.0% Coverage on 45 New Lines to cover

0.0% Duplications on 632 New Lines

Since we certainly don't want to increase the already large debt of this system, we addressed the code smells and got the following results.


New Code
Since April 20, 2021
Started last month

Overall Code

0 New Bugs — Reliability A

0 New Vulnerabilities — Security A

0 New Security Hotspots — Reviewed — Security Review A

0 Added Debt — 0 New Code Smells — Maintainability A

0.0% Coverage on 60 New Lines to cover

0.0% Duplications on 670 New Lines

As can be seen on the screenshot, we introduced our ranking system without any added technical debt, vulnerabilities, bugs and duplication. However, both our computers did not have enough memory to include the JaCoCo XML test report (uploaded to the GitHub repository) in the SonarQube analysis (even with a 3gb Computer Engine, 3gb ElasticSearch and 4gb general java heap increase). We did cover every added line with tests, and validated that with the Intelij Coverage functionalities.

Given that the tests could not be imported, the improvement we made to the system will not be reflected in most of the SonarQube reports. But we can confidently say that the things we added, did not hurt the system in any way, as we increased coverage and tested each function we touched.

9 Conclusion

Reengineering MegaMek all started by getting to know the game itself, which turned out to be more complicated than expected (neither of us were particularly good at it). The next step we took was exploring the code. The codebase is huge, so this took quite some time to both delve through it, but we complemented each other when one was not sure about something. After getting to know the code, we already had some ideas on where and how the ranking system could be accomplished.

Next we started running our reengineering toolset on the project. We identified problems such as duplicated code, high complexity functions and missing tests. Throughout all this our understanding of the system kept growing, which led us to specifying a more and more refined redesign.

After having a set redesign, the search for refactoring targets began. Due to the huge size of classes such as Server, we had quite a bit of trouble with this. The enormous size overwhelmed us and our refactoring targets were a bit too broad. More importantly, we were unsure if they would be fruitful. To progress, we made a simple prototype to narrow down the scope. This worked well, so well that we decided to refine this implementation and eventually merge it to the main branch. During this process we focused our refactoring and testing to the added and touched functions in our implementation. Because all the touched functions had a quite decent code quality, testing was the main focus.

When looking back at our reengineering journey, we feel some things could be improved in our process. The search for refactoring targets was a slow one, and if we would have started prototyping quicker there would have been more time to spend on refactoring and testing all the parts around our ranking system. Moreover, we learned the importance of staying in scope. At first we were quick to jump at large code duplications, especially when we felt we could improve the code quality. However, after deciding to focus only on things in scope, we were able to progress much quicker and even come to a basic and fully tested implementation.

The ranking system was only partially achieved. UI, persistence and a ranking algorithm such as ELO are parts that are missing, with a persistence design being suggested. Refactorings were also limited to our scope. However, in case the whole project would be refactored, we have made suggestions on how to refactor the god class Server, which would drastically increase the quality and reliability of MegaMek.