INF3121/4121
Project Assignment 1

Esben Slaatto - esbenss
Camilla Stenberg - camilest

link to Github-repo:
https://github.com/esb1/inf3121

Requirement 1:

BRIEF DESCRIPTION OF THE PROGRAM:
We have chosen the project «8.Hangman-Java» for the assignment. The program is a simple version of the commonly known hangman game, written in Java. The objective of the game is to guess a hidden word, one letter at a time. In addition to guessing letters and playing the game, the program provides functionality for restarting the game, exiting the game, help (cheat). It also provides the opportunity to store names on a scoreboard. The high-score list will be stored also when the program is exited and restarted.

TEST DESIGN:
Based on the description above, we have created a brief overview of the different user choices for the game:

- Game started, 5 choices:
  1. Restart : Restart the game
  2. Exit : Exit the game
  3. Help : Get help with guessing
  4. Top : Show scoreboard
  5. Other input / Play game:
     a. Correct letter guessed
     b. Uncorrect letter guessed
     c. Invalid input from user

For designing the tests we used a combination of different black-box testing techniques. Mainly we used a combination of **state transition testing** and **error guessing**. In addition we chose to base the tests cases for the letter-guessing input on **equivalence partitions,** to make sure we covered all possible errors**.**
For the state transition tests we used the list above to design the tests. We decided that all of the 5 choices the user had the program would change it's state in some way, and all of these ways needed to be tested.
When creating the test cases for bullet point 5 from the list we decided on three partitions of input, valid - both correct and incorrect, and invalid.
Previous knowledge helped us design tests based on the error guessing technique. We know that invalid input is a common mistake made by users. Therefore, we choose to test for different types of invalid input such as special characters that many systems is failing to handle in a good way.

We have partitioned the tests into three groups:
- Menu functionality tests
- Input tests
- Game functionality tests

The test steps for each of the test cases will look quite alike. An example of the test steps for the test case «restart» are provided below:
1. Decide on expected result
2. Run program
3. Enter «restart»
4. Press Enter
5. Compare expected result to actual result
6. Give test case a verdict, pass or fail

Table showing all tests with test cases and respective conditions and results

Test	Test Case	Pre Conditions	Post Conditions?	Expected Result	Actual Result	Priority
Menu functionality	Restart	Game running, waiting for input		Game restarted		9
	Exit	Game running, waiting for input		Game exited		8
	Help	Game running, waiting for input	Set flag: Help used	One letter revealed		10
	Top / Show scoreboard	Game running, waiting for input		Scoreboard shown		11
Input tests	One letter	Game running, waiting for input	Test letter against secret word	Game accepting input.		1
	Special chars.	Game running, waiting for input	Ask for input	Invalid input message		2
	Multiple characters	Game running, waiting for input	Ask for input	Invalid input message		3
Game functionality	Correct unguessed letter	Game running, waiting for input	Ask for input	Reveal all instances of guessed letter in secret word.		4
	Correct previous guessed letter	Game running, waiting for input	Ask for input	No action		12
	Non-correct letter guessed	Game running, waiting for input	Update num-guesses-counter.	Display "wrong guess" message		5
	Word guessed, no help	Game running, waiting for input	Scoreboard updated.	Win message + scoreboard updated		6
	Word guessed, help used	Game running, waiting for input		Win message		7

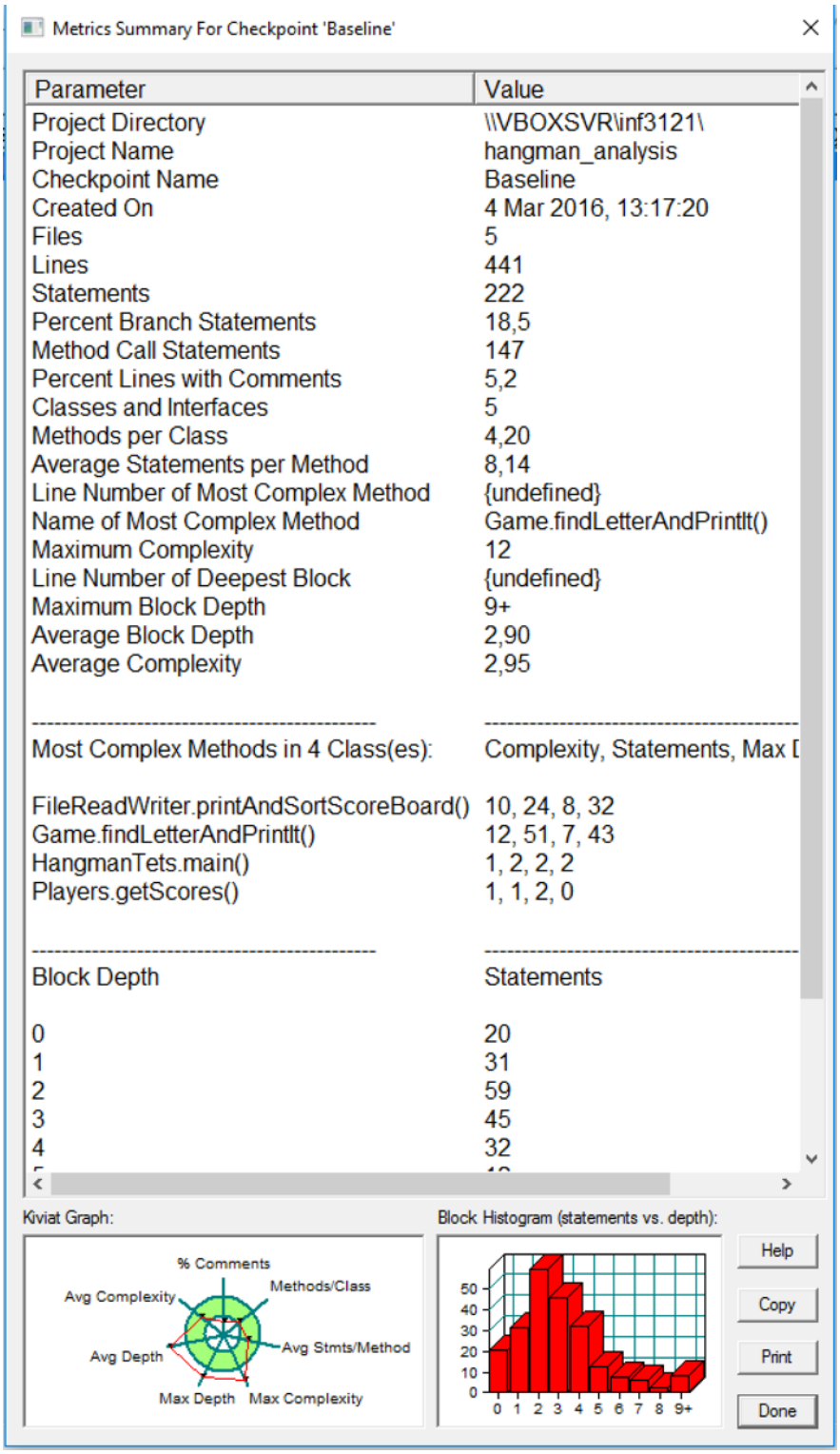
NON-FUNCTIONAL TESTING:

In this project it would make sense to do some non-functional testing such as usability and maintainability testing. Usability testing is important because the program is made for a user to interact with it. Maintainability testing is almost always important, because you should always assume that the code should be updated, maintained or tested throughout its life cycle, because errors can be detected after the release, or the program needs improvement for some reason.

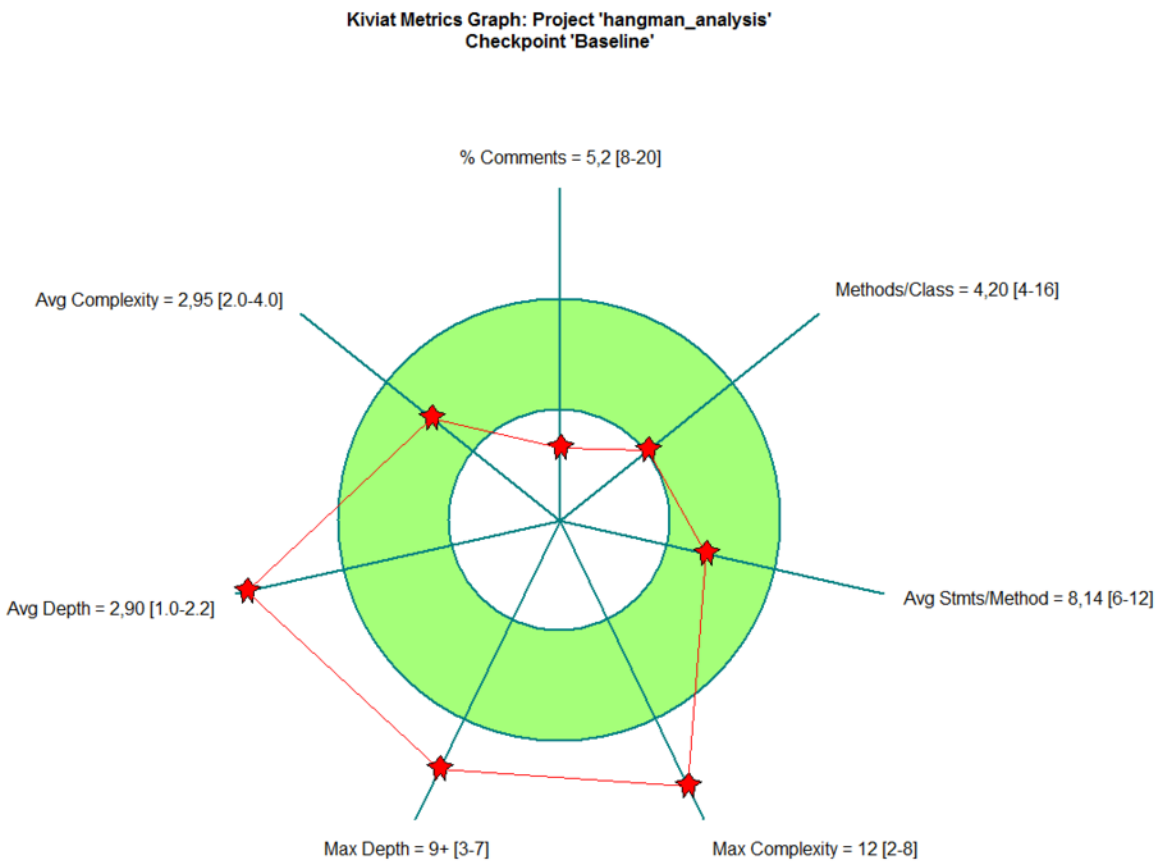
For this project, it does not make much sense to prioritise non-functional tests for speed, stress and load. This is because of the size and expected use of the program, and it is not intended for multiple users or processes.

Requirement 2:
Metrics at project level:

A screenshot of the metrics summary for the project Hangman in the state it was when downloaded,



Screenshot of the Kiviat Metrics Graph for the project Hangman in the state it was when downloaded.



Below you find a brief description of metrics displayed in the Metrics Summary for Checkpoint view in SourceMonitor for the project Hangman.

GENERAL METRICS:

- Files
  - Number of source files in system.
- Lines
  - Total number lines of code in source files.
- Statements
  - A statement is an instruction followed by a ';', an attribute, or a branch (if, else, while etc…). The metrics counts total number of statements in project.
- Percent Branch Statements
  - Calculates the part of statements being branch-statements.
- Method call statements
  - Total number of method calls in project.
- Percent Lines with comments
  - How much of the source code is commented with standard comment-syntax. Counts both descriptive comments, and code blocks being commented out.
  - The value is 5,2 and expected range is from 8-20. This value is lower than expected in the static analyser tool, but it should be closer investigated to find out if it is too few comments.
- Classes and Interfaces

    - Total number of classes and interfaces in project. Inner classes, both anonymous and not are also counted.
- Methods per class
    - Total number of methods divided with total number of classes on project level.
    - The value is 4,20, and expected range is from 4-16. This means that the value is quite low compared to the expected, but still within the "approved range".
- Average Statements per Method
    - Total number of Statements divided on total number of methods.
    - The value of this metric is 8,14 and it's just as expected, within the expected range from 6 to 12.

COMPLEXITY METRICS:

- Four metrics is displayed for complexity:
    - Maximum complexity
        - Only the complexity of the most complex method is shown. The complexity metric is the number of executional paths through a method. (Every branch statement adds a count to complexity)
        - The value of the most complex method in the project is 12, and this is a lot higher than the expected range from 2 - 8.
    - Name and line number of the most complex method is displayed in the metrics summary as well.
    - Average complexity
        - The sum of all methods complexity divided by the total number of methods.
        - The value is 2,95 and is right in the middle of the expected values ranging from 2 to 4.

BLOCK DEPTH METRICS

- Maximum block depth
    - Calculates the maximum number of nested statements among all of the source code. If more than 9 levels of nesting is found 9+ is displayed.
    - The value of the most nested statement is higher than 9, and this is much higher than the expected range going from 3 to 7.
- Line number of deepest block
    - The line number of
- Average block depth
    - The sum of all statement's nesting-level divided by all statements.
    - The value is 2,0, and this is higher than exptected. Expected values should be between 1,0 and 2,2.

After observing and analysing the metric's expected and actual values, it looks like the most important to improve are the maximum and average block depth. Maximum complexity values should probably also be improved. Percentage of comments are quite low and should be taken a closer look at, to see if the source files needs more or better documentation.

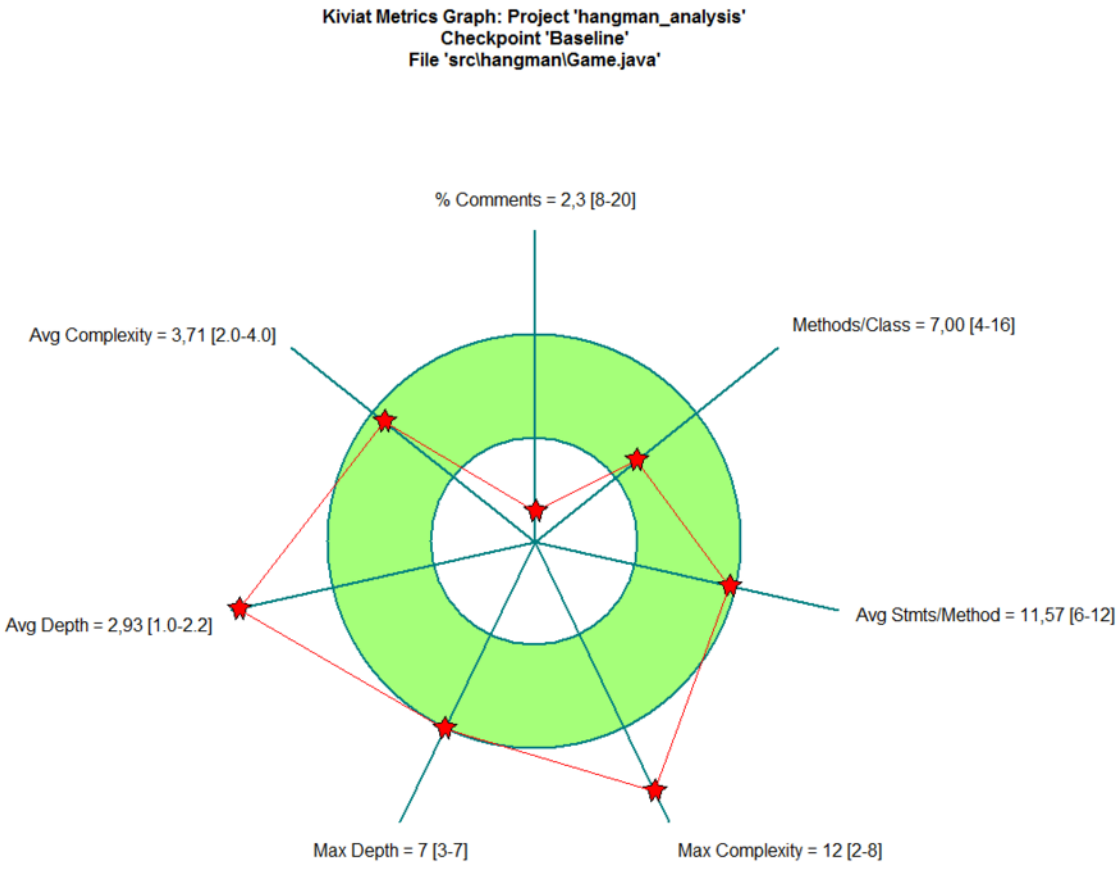
FileReadWrite.java is the biggest file in the project with 219 lines of code.

The file 'FileReadWrite.java' is the file with most branches in the project with 20,8% of the statements being branch statements.

The file 'Game.java' seems to be the file containing the most complex code.
It has the highest value when looking at both, average complexity and maximum complexity. We think that both of these metrics is telling something useful about the complexity of the code, and both should be taken into consideration when ordering files according to complexity.

Metrics at file level

Below is a screenshot of the Kiviat Graph of the file Game.java from project Hangman in the state it was when downloaded.



In the file Game.java, which contains most of the games logic, there are a few metrics that are far outside the expected range. The percentage of comments is a lot lower than expected, while average depth and maximum complexity has far higher values than expected. Average statements per method, average complexity and maximum depth is on the high end of the expected range, while methods per class is right in the middle.

When we compare the metrics displayed in the kiviat graph for both the file Game.java and the source files all together we can see that there is a lot of similarities. The values that are high for the entire project are also high in Game.java, and the same with the low values. Some are more extreme when just looking at the one file, for example the comments metric, while the max depth is closer to expected than for the project.

We would definitely refactor some of the methods in Game.java to try to get the metrics closer, or within, the expected range. The best action could be to identify the blocks that increases the avg depth, and also look at the method with the highest complexity. It might also be a good idea to look at the comments across all the methods.

Requirement 3

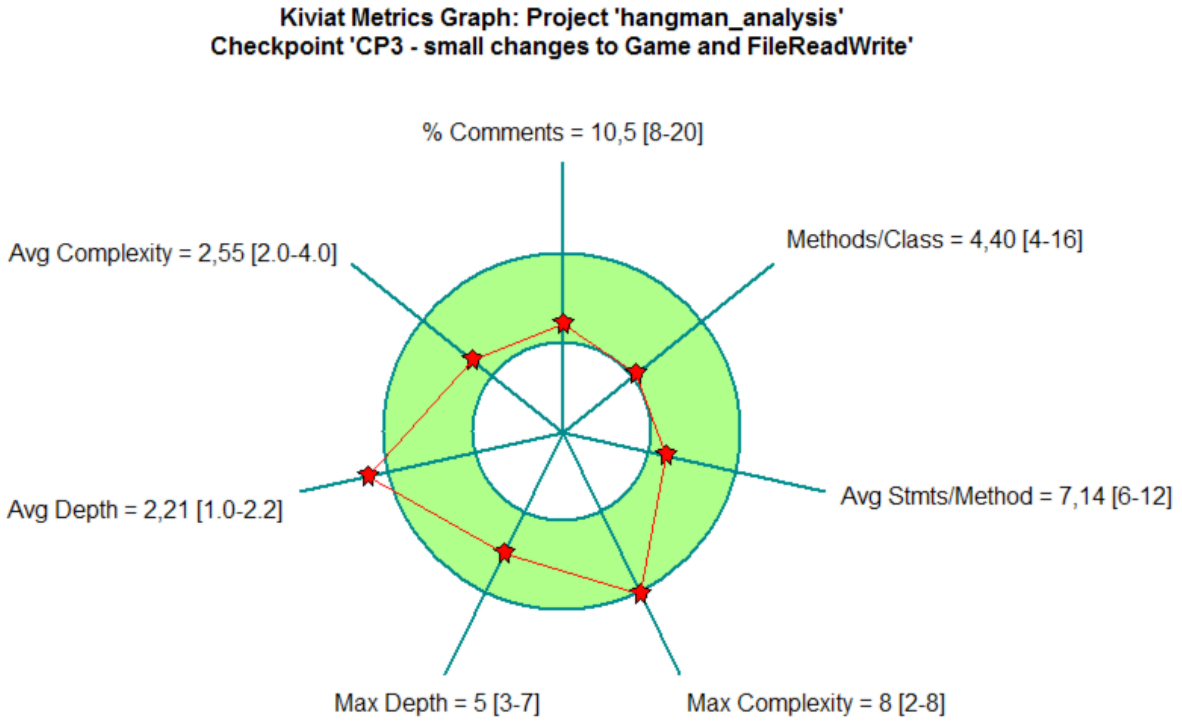
Metrics at project level:

In requirement 2 we identified on a project level two major metrics that needs improvement, complexity and block depth, in addition to the 'percentage of comments' metrics that needs to be considered improved as well after looking at the source-code. We find it difficult to decide wether there is sufficient documentation in the code just by looking at the static analyser tool, so we will improve the comments if needed when trying to improve the other two metrics.

We found that when analysing the values in Sourcemonitor that we would start to refactor the two main files in the project, Game.java and FileReadWriter.java as these files seemed to be the ones making the values in complexity and depth metrics outside the expected range.
After changing the code in these two files we found that the metrics improved in such a way that we did not spend time on the remaining three files.

We have uploaded the modified project to GitHub, and here are two links, one to the source-files, and one to the root folder of the project. The only files changed from the original project is in the source-folder.
Link to all sourcefiles in Hangman-project:
https://github.com/esb1/inf3121/tree/master/oblig1/8.Hangman-Java/Hangman-Java/src/hangman
Link to the root folder of Hangman-project:
https://github.com/esb1/inf3121/tree/master/oblig1

Screenshot of the Kiviat graph at project level after we modified the to major files in the program.



Most of the changes in the code has been removing redundant or unnecessary code, but we did also extract some code from the most complex method into additional methods, and tried to "simplify" some of the most nested code-blocks where we found unnecessary nesting.

There are still a few unused methods in one of the files, but in lack of comments in these methods it is difficult to know exactly why they are there. Instead of deleting them, we just improved the code and left them in place. They don't add a lot of complexity or anything to the project, and they can be removed if it's latere decided that they'll never be used.

The kiviat graph of the entire project after modification can be seen above, and after analysing the values we can see that the three "worst" metrics has improved quite a bit.

- Maximum depth has gone down from over 9 to 5
- Average depth is still not within the expected range, but getting closer after being decreased from 2,9 to 2,21
- Maximum complexity ended at 8 after being as high as 12 before the refactoring.

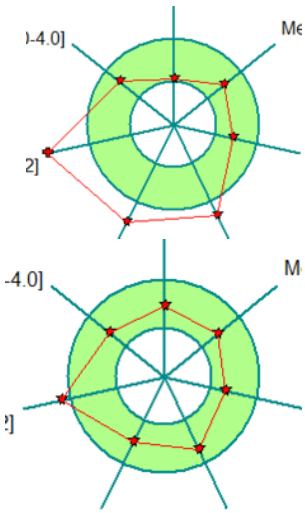
In addition to this we did add some comment lines to the source code. Most often it was just a line briefly explaining the following method, or clarifying a codeblock. But we read from the graph that this was more than enough to bring the metric 'percentage comments' within the expected range. If we were to comment the entire project with JavaDoc style commenting, it appears that the value will peak far over the expected range, which tells us that the metric regarding comments can only be used as guideline, and that the development team needs to lay the ground rules as to how they want to document their program.

Metrics at file level:

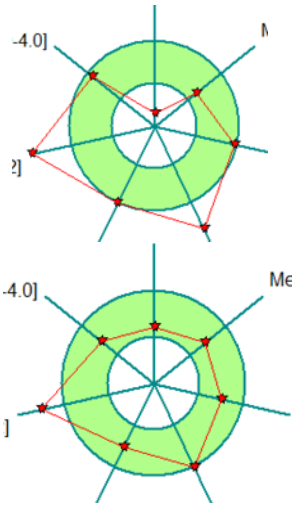
At file level we see from the new metric values that the two files we modified both improved in the same places as we saw improvement on the project as a hole. This is not very surprising, as these two files contains so much of the code in the project. About 350 out of 400 lines of code is found within these two files.

In both of the modified files we did manage to improve max complexity, max block depth and average block depth to more acceptable values. The comment metric also improved a little. This was more of a "furtunate side effect" and happened probably just as much thanks to removal of code lines as to addition of comments, even though documentation of the code was also improved.

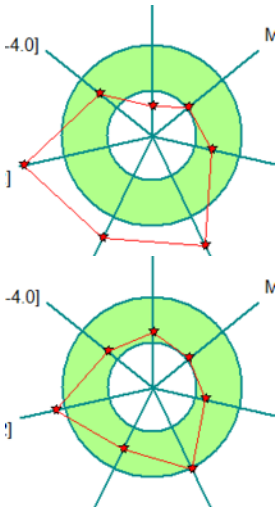
Metrics for the file FileReadWriter.java before and after refactoring.



Metrics for the file Game.java before and after refactoring.



Metrics for entire project before and after refactoring and improvement



It did take us some time to understand the code well enough to find good ways to change it. There were some code blocks and methods that would have been easier to get a grip around had there been a few more comments.
We did get some help from our IDE as to find the usage of some methods, as this is the most difficult thing to find by just looking around. The fact that there were a bit of unnecessary parts made understanding the code slightly more difficult as well.
On the upside, there were most often quite descriptive variable- and method names which helped the general understanding of methods despite lack of a few comments.