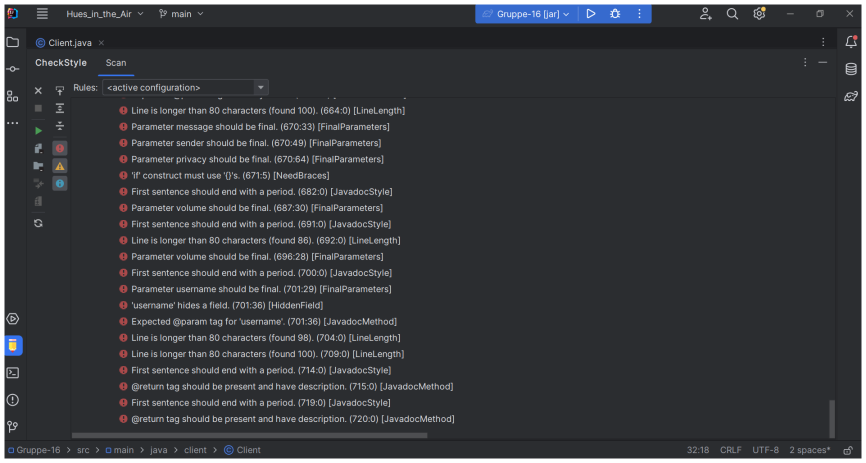
## 1 Measures

In this document, we are outlining our approach we took to ensure that the final product of our project is of the highest possible quality. As a team of four, we understand that any project of this nature can be prone to errors, bugs and glitches. Therefore, it was our goal to identify and mitigate these issues early on in the development process, to ensure that we deliver a smooth and enjoyable game experience to our users. The following is an exhaustive list of the steps we took to ensure the final product is robust, reliable, and bug-free.

- Defining clear requirements: Before starting the development, we defined clear and specific requirements for the game. These requirements covered all aspects of the game, from its gameplay mechanics to its visual design and user interface. Knowing what we would be planning on programming in the near future allowed us to organise our code in an appropriate way: it was then easier to refactor or modify to implement features later on.
- Conducting thorough testing: We conducted extensive testing throughout the development process, including unit testing, integration testing and user acceptance testing (exterior opinions and suggestions). These tests covered critical, as well as non-critical parts of our code whenever we saw the need for it. We built our tests with JUnit tests, and the code coverage we measured with Jacoco.
- Whenever they arose, major bugs were reported on either discord in a dedicated text channel or on GitHub. Each of those bug reports contained the following information:
  - Commit hash
  - Operating System / Environment
  - Description of the problem
  - How to reproduce it
- At the beginning, we split up in groups of two to program the server and the client. Later on, one group focused on the menu while the other one on the game rules of the project. In each pair, both developers kept their colleague updated on their progress, explaining their code and what it does. After having code explained to them, team members reviewed the code and gave feedback to improve it.
- Documenting the code: We documented our code thoroughly with Javadoc, making it easier for each and every one of us to understand how the code works and to identify any issues or bugs if they are to arise. We are aware that at the beginning of the project we didn't wrote enough thereof. However, since milestone 3 we payed attention to document the program enough and accomplished this achievement.
- Additionally, we set some standards for our code, in order to keep it as clean, concise and efficient as possible:
  - Write short methods. If the code gets to hefty, redistribute in sub-methods that can be reused from other places in the code.
  - Use Javadoc before and within methods as much as possible to make the code more understandable.
  - IntelliJ allows us to see the number of method calls of each method as well as the number of usages of variables. We used this tool to estimate how meaningful our methods are.
  - We used Jacoco and 3 other metrics to evaluate how well we have implemented the previous points and modified the code if they report inconsistencies or illogical choices. We will develop this aspect in the section 5 of this document.
- We used the logger library Log4j2, to help us debug our code and save the outputs in log files.



```
Line is longer than 80 characters (found 100). (664:0) [LineLength]
Parameter message should be final. (670:33) [FinalParameters]
Parameter sender should be final. (670:49) [FinalParameters]
Parameter privacy should be final. (670:64) [FinalParameters]
'if' construct must use '{}'s. (671:5) [NeedBraces]
First sentence should end with a period. (682:0) [JavadocStyle]
Parameter volume should be final. (687:30) [FinalParameters]
First sentence should end with a period. (691:0) [JavadocStyle]
Line is longer than 80 characters (found 86). (692:0) [LineLength]
Parameter volume should be final. (696:28) [FinalParameters]
First sentence should end with a period. (700:0) [JavadocStyle]
Parameter username should be final. (701:29) [FinalParameters]
'username' hides a field. (701:36) [HiddenField]
Expected @param tag for 'username'. (701:36) [JavadocMethod]
Line is longer than 80 characters (found 98). (704:0) [LineLength]
Line is longer than 80 characters (found 100). (709:0) [LineLength]
First sentence should end with a period. (714:0) [JavadocStyle]
@return tag should be present and have description. (715:0) [JavadocMethod]
First sentence should end with a period. (719:0) [JavadocStyle]
@return tag should be present and have description. (720:0) [JavadocMethod]
```

Hues_in_the_Air
main
Gruppe-16 [jar]
Client.java
CheckStyle
Scan
Rules: <active configuration>
Gruppe-16 > src > main > java > client > Client
32:18 CRLF UTF-8 2 spaces*

Figure 1: The checkstyle errors before milestone 3

## 2 Checkstyle

Being four to write code in the project it appears fundamental to have the same style of programming, especially when it comes to read and understand the code which was written by someone else. It makes it way more readable if everyone uses the same conventions. Therefore we all agreed on using the same writing style with the google-conventions.

We only introduced this tool shortly before milestone 3, as the program became more extensive. Therefore it looked quite improvable as shown in Figure 1, but it got better as soon as we had revised the whole code and we got used to the conventions. We expected that by the end of this project, we would have a homogeneous written code. This is actually the case, as for example the lines do not exceed 80 characters and there is enough and better structured Javadoc. In retrospective, we learned a lot about the style of our coding and are more aware of the way to write the code.

## 3 Jacoco and JUnit tests

As already mentioned before, we used Jacoco to measure the code coverage and to facilitate the seek of bugs and errors. However, it is not the only tool regarding quality assurance. It will also help us finding code unconventionally written so that we can reevaluate that flaws. We also want to use JUnit tests to make sure our methods work properly and to find bugs and errors more easily.

## 4 Logger

Additionally to the tools mentioned before, we used loggers from the logger library log4j2 in our project. This helped us finding our errors more easily, as it doesn't tell us only the message but also the importance of the outcome (error, warn, info,...). We incorporated 88 loggers in our project.

## 5 Three other metrics

As it is required for the project, we incorporated 3 other metrics by means of metrics reload plugin from intellij. For each metrics we took the data for the methods and classes separately, and measured the minimum, maximum and average.

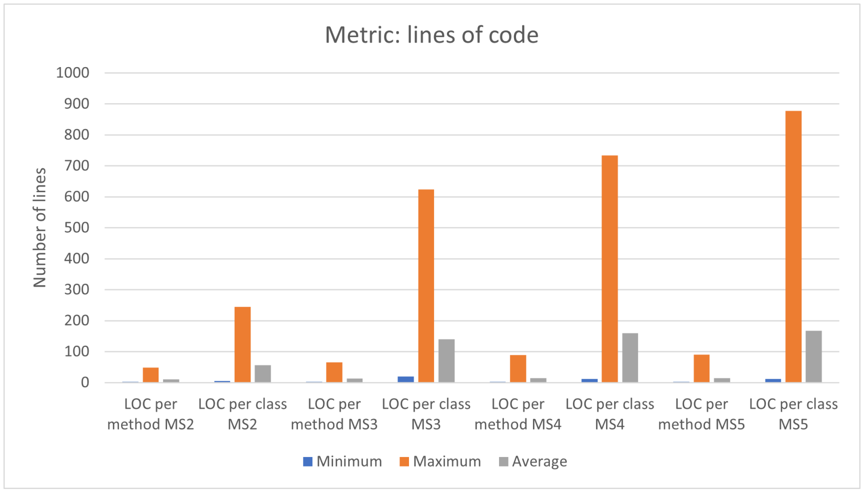


Figure 2: Lines of code per method and class from metrics reloaded plugin

## 5.1 Lines of code

To start of, we measured the amount of lines of code as shown in the figure 2. At the beginning of the project, we had a manageable number of lines per method and class. As expected, the code became larger and more complex as we went on with the project. Especially between the second and forth milestone, the number drastically increased. Becoming aware of the rising number, we agreed that the number of lines per method should not exceed 100. Being at the end of the project, we are satisfied as they stayed in that range. Regarding the number of lines per class we are aware that the maximum lines of code is quite large. However, the average number of lines per class is 167. This shows, that most classes are in an acceptable range. For a future project we intend to pay more attention since the beginning to keep the code more manageable by rather writing more classes than have many lines of code.

## 5.2 Complexity metric

The second metric we used is the cyclomatic complexity metric which tells us how complex the code is written. In this metric we had more difficulties to stay in the range that we set as a goal. As represented in figure 3 we exceeded the number 20 as cyclomatic complexity of methods (CYC of methods) since milestone two. We are aware, that 20 is commonly an upper bound for the complexity of a program. However, we didn't manage to reduce this number. As we saw that it was getting impossible to simplify the code we had written until then, we decided to set a new upper bound of 25 until the end of the project. Unfortunately we also exceeded this number by two, but as it was only on milestone 5 we agreed on leaving the code like that. For a next project however, we will try to stay in the bounds we set since the beginning, so that the code is more understandable. Regarding the weighted method count (WMC) it got also hard to keep the number in a acceptable range as the methods were yet complex. However we almost managed to stay under 100 until the end of the project, which was our goal.

## 5.3 Dependency Metric

As last metric we chose the dependency metric which shows us the number of dependencies (Dcy) and dependents (Dpt) per class as seen in figure 4.

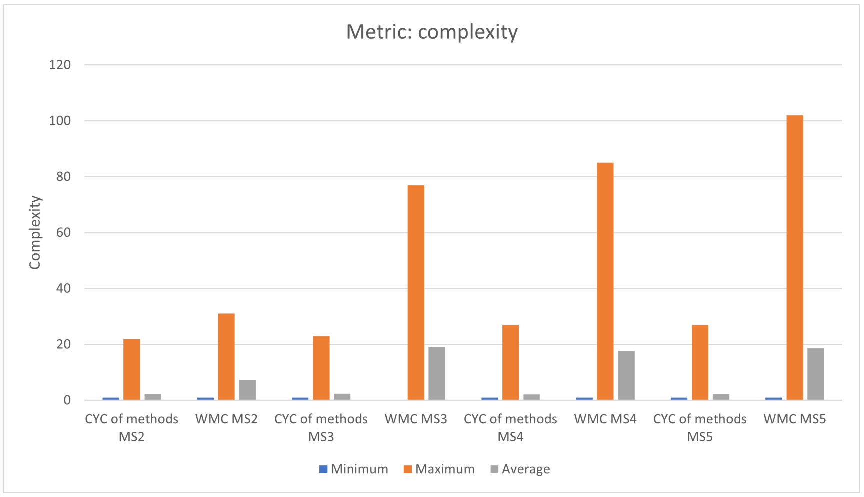


Figure 3: Complexity metric from metrics reloaded plugin

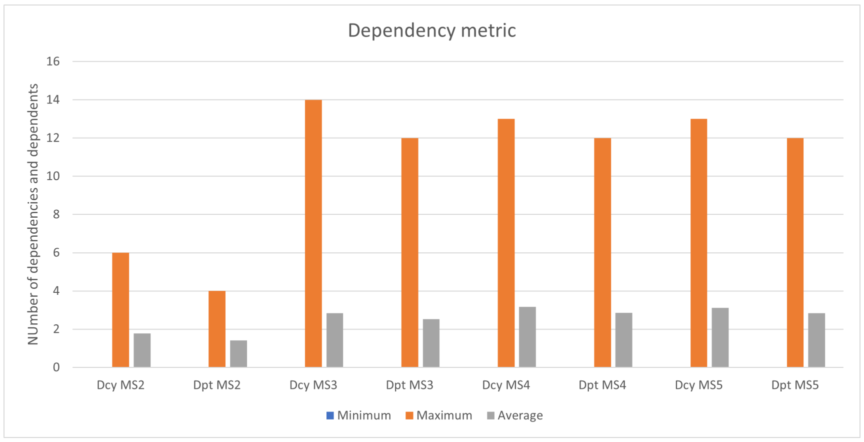


Figure 4: Dependency metric from metrics reloaded plugin