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 we measured the code coverage with Jacoco. The results of this two tools are discussed in section 3.
- 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.

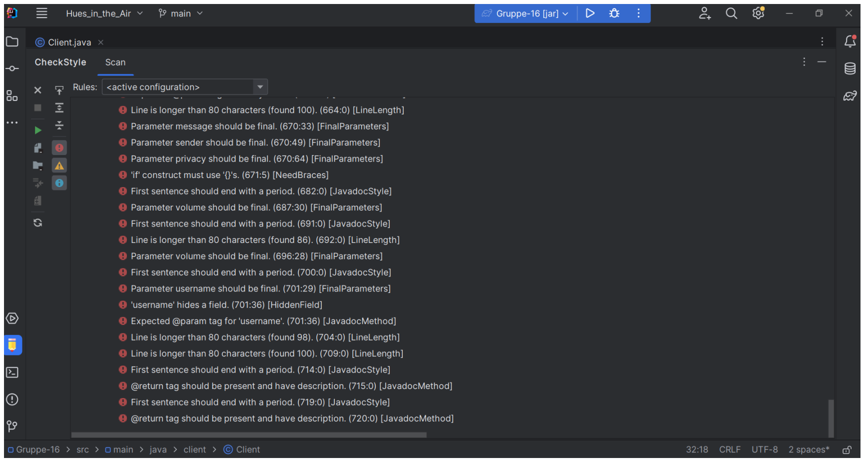
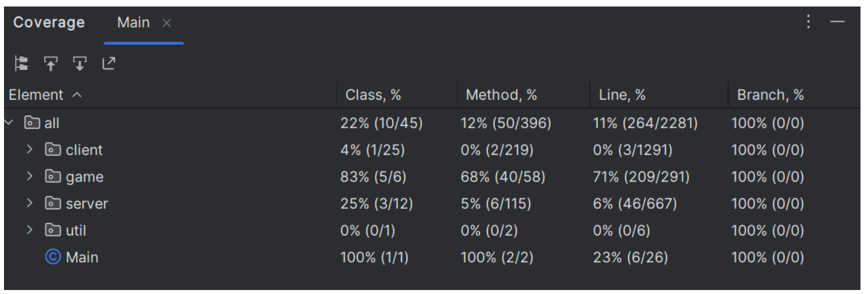


Figure 1: The checkstyle errors before milestone 3


Coverage Main

Element	Class, %	Method, %	Line, %	Branch, %
all	22% (10/45)	12% (50/396)	11% (264/2281)	100% (0/0)
client	4% (1/25)	0% (2/219)	0% (3/1291)	100% (0/0)
game	83% (5/6)	68% (40/58)	71% (209/291)	100% (0/0)
server	25% (3/12)	5% (6/115)	6% (46/667)	100% (0/0)
util	0% (0/1)	0% (0/2)	0% (0/6)	100% (0/0)
Main	100% (1/1)	100% (2/2)	23% (6/26)	100% (0/0)

Figure 2: Results of the Jacoco metric

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 JUnit tests and Jacoco

We used JUnit tests to make sure our methods worked properly and to find bugs and errors more easily. With them, we tested the main components of our game, which mostly consists of the cube and it's actions. For instance, if the acceleration is set to the right angle during and after a jump or if the collisions are detected and handled properly.

As already mentioned before, we used Jacoco to measure the code coverage and to facilitate the seek of bugs and errors. As seen in Figure 2, the most tested package is game, as it contains the rules of the game. We are satisfied about the 83 per cent of code coverage, as our goal was to reach 80 so that most of the class would be tested, but all of it wasn't necessary.

Metrics Reload plugin intellij			
MS2	Minimum	Maximum	Average
LOC per method MS2	3	49	10,54
LOC per class MS2	5	244	56,23
CYC of methods MS2	1	22	2,29
WMC MS2	1	31	7,23
Dcy MS2	0	6	1,77
Dpt MS2	0	4	1,41
MS3	minimum	Maximum	Average
LOC per method MS3	3	66	13,09
LOC per class MS3	20	624	140
CYC of methods MS3	1	23	2,4
WMC MS3	0	77	19,1
Dcy MS3	0	14	2,83
Dpt MS3	0	12	2,52
MS4	Minimum	Maximum	Average
LOC per method MS4	3	89	14,09
LOC per class MS4	12	734	160,09
CYC of methods MS4	1	27	2,1
WMC MS4	1	85	17,71
Dcy MS4	0	13	3,17
Dpt MS4	0	12	2,86
MS5	Minimum	Maximum	Average
LOC per method MS5	3	90	14,46
LOC per class MS5	12	877	167
CYC of methods MS5	1	27	2,17
WMC MS5	1	102	18,67
Dcy MS5	0	13	3,11
Dpt MS5	0	12	2,83

Figure 3: All metrics data

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,...) and at which moment the errors occurred. A major advantage to use loggers was that the information is given in a separate file which makes it more readable as in the console. 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 reloaded plugin from intellij. For each metric we took the data for the methods and classes separately, and measured the minimum, maximum and average. In Figure 3 every data is represented in an excel table. In the following paragraphs we are going to analyze each metric separately.

5.1 Lines of code

To start of, we measured the amount of lines of code as shown in Figure 4. 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 with a percentage of 65. Becoming aware of the rising number, we agreed that the number of lines per method should not exceed 100. We set this number because otherwise the code would become illegible and we noticed that it was useless to have more lines of code. 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

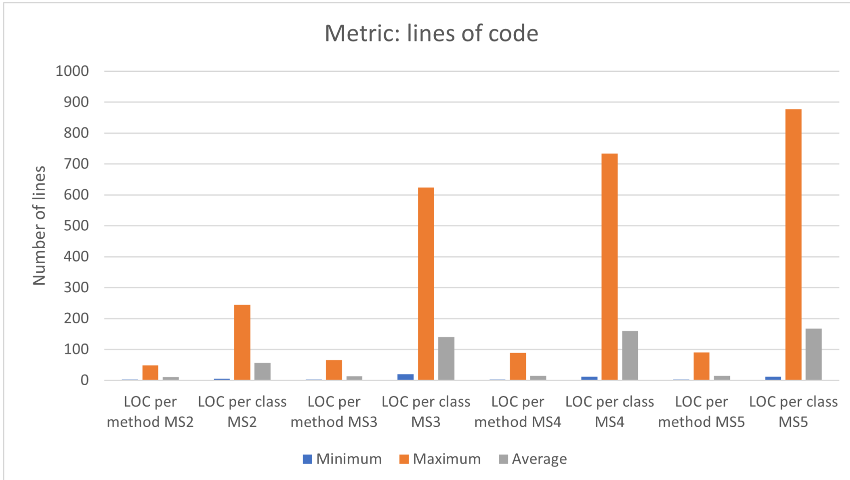


Figure 4: Lines of code per method and class

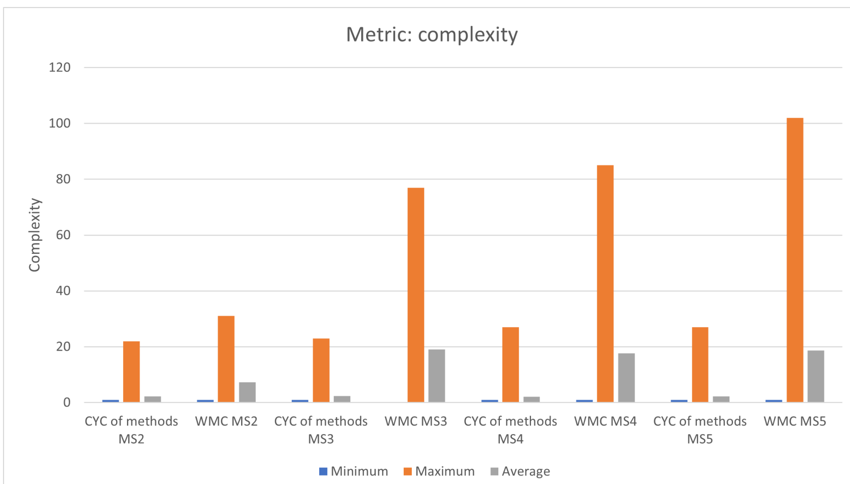


Figure 5: Complexity metric

are in a lower 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

Cyclomatic complexity is a source code complexity measurement that was developed by Thomas McCabe in 1976 to which correlates a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of independent paths through a program module. As represented in Figure 5 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, as we are not yet professionals and our time to code was limited, we didn't want to set the goal too high. We therefore agreed that we didn't want to exceed the number 25, what we achieved until milestone 5. Regarding the weighted method count (WMC) it got also hard to keep the number in an 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.

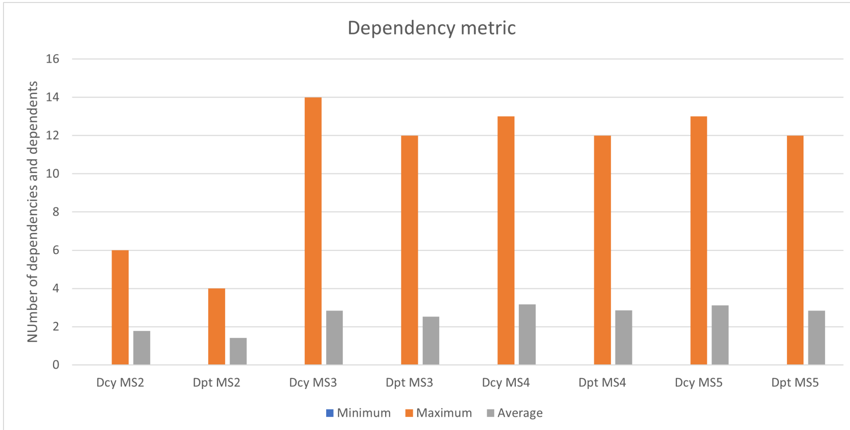


Figure 6: Dependency metric

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. In Figure 6, it's noticeable that between milestone two and three there has been a big step up in the number of dependencies. This is mostly due to the project getting momentum, so the complexity of the code increased. However we managed to keep the number of dependencies in a manageable range: in milestone 5 we are at thirteen dependencies for the maximum, which we are satisfied about. For the number of dependents we are also content about our job, as it didn't exceed 12. For both of them we know that our upper bound was not set really high, but as our time to code was limited and we are not yet professionals, we are proud of us. Moreover, in average we stayed between the number two or three, which is a great job.