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, user acceptance testing (peer review but also critical feedback from colleagues) and some intuitive integration testing. 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, bugs were reported immediately to the other developpers of the group. In the beginning of the project, we would share them in a dedicated text channel on discord with the following information:
  - Commit hash
  - Operating System / Environment
  - Description of the problem
  - How to reproduce it

  However, we later got to know Stepsize, an online issue tracker which enabled us to create issues which would link directly to the faulty piece of code. The gravity of the issue could also be given, choosing from “blocker”, “morale”, “quality”, “security” and “velocity”. Issues could then be commented on, discussed and resolved much more easily than with discord.
- At the beginning of the project, we split up in groups of two in order to program the server on one side and the client on the other. Later on, one group focused on the menu while the other one on the game rules of the project. Splitting up in group that way enabled us to work on independent parts of the code base separately, therefore making faster progress. During this whole process, communication was key: in each pair, both developers kept their colleague updated on their work, explaining their code, what it does and why it does it. 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 without further discussion with other group members.
- 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 too 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 and to remove unused variables or methods as soon as we saw them.

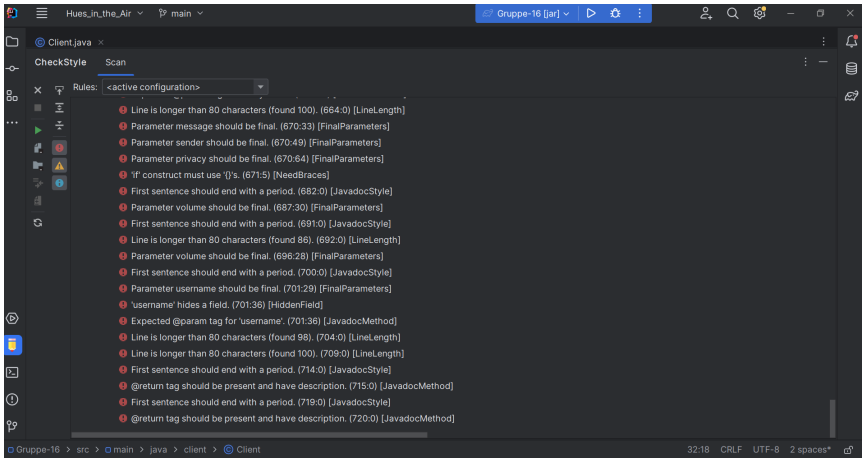

Figure 1: The checkstyle errors before milestone 3

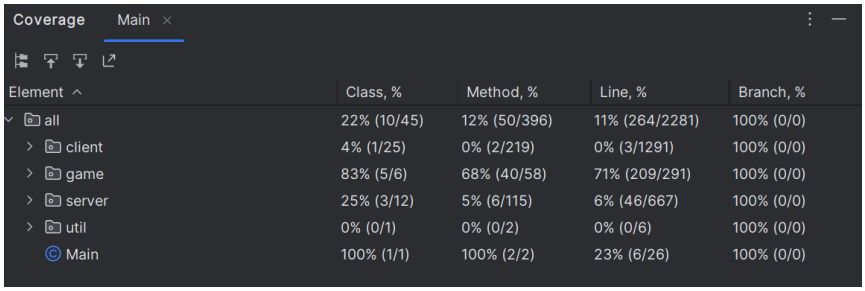

Figure 2: Results of the Jacoco metric

  - 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. This was arguably more efficient than printing out statements to the console because we were able to access these errors in a separate text file which contained much more information than a standard terminal: the gravity of a statement (error, warning, info) and the exact moment at which it was outputted.

2 Checkstyle

Being four to write code in this project, it appeared fundamental to have the same style of programming, since this is vital in order to facilitate the reading and further developing of code which one hasn't written. This might seem like a very subjective goal to attain, we therefore agreed to use the google conventions to standardise our code. This was made easier by the use of a plugin in IntelliJ which verified whether these conventions were being respected.

We only introduced the checkstyle plugin shortly before milestone 3, as the program became more extensive. Therefore, despite having a lot of errors when first running the tests (1), we managed to greatly improve our code quality by going through the errors thorougly and getting used to some new conventions.

We expected that by the end of this project, we would homogeneously written code. We are confident that this is the case, since Javadoc does not produce any errors and coding conventions are respected everywhere. In retrospective, these are the two things that we learned most about.

3 JUnit tests and Jacoco

We used JUnit tests to make sure our methods worked properly and to find bugs and errors more easily. They mostly helped us in the testing of the main components of our game, which englobes the cube movement and collisions for the most part. These tests allowed us to find certain inconsistencies in the gameplay, however, they are now all green.

As already mentioned before, we used Jacoco to measure the test coverage of our code. As seen in Figure 2, all components of the program are not tested, as we mostly concentrated on the game package. This was a very conscious decision we made, since we know that that package is where bugs would come from, but also where they would be the most felt. We could not allow ourselves to have some faulty gameplay.

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 told us not only the contents of messages but also the their importance (error, warn, info,...) and at which moment they had been created. A major advantage of using loggers was that the information is given in a separate file which made it much more readable than a standard console. In total, 88 logger statements are present in the project, both on the server's side and on the client's side. This was more than enough to inform us of the exact status of the client and the server at every moment in time, but also not too much, so the log files weren't flooded with useless information such as messages sent in the chat or the cube's position.

5 Metrics

As required for the project, we studied three metrics with the help of the 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. This can all be seen in Figure 3. In the following paragraphs, we will be going more into detail of what was seen, changed and learned from each metric.

5.1 Lines of code

To start off, we measured the amount of lines of code as shown in Figure 4. This number gradually increased during the project, making a big leap between milestones two and four, since this is about when we started implementing FXML in our app.

Being aware of this rising graph, we set agreed that the number of lines of code per method should not exceed 100. We set this number for several reasons: bigger methods are harder to read, harder to modify, more prone to error, and generally avoidable anyways. At the end of the project, we managed to stay under that threshold, which we are satisfied with. The number of lines of code per class reaches a great maximum since our Client class contains all the connections between the app, the different controllers and the game -which was unavoidable-, our average of 167 is however something we find reasonable.

Had we had more time available, we would have concentrated on making more concise classes and maybe applying more of Java's OOP features in order to limit the size of the Client and Server classes. This is something that we would surely concentrate on for future projects.

5.2 Complexity metric

Cyclomatic complexity is a source code complexity measurement that was developed by Thomas McCabe in 1976 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. Whether the cyclomatic complexity of our code is good or bad is arguable, since different sources

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

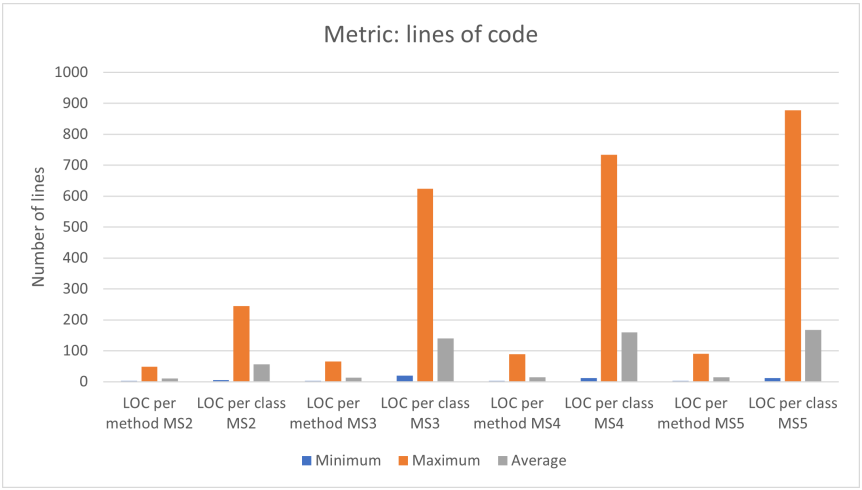

Figure 4: Lines of code per method and class

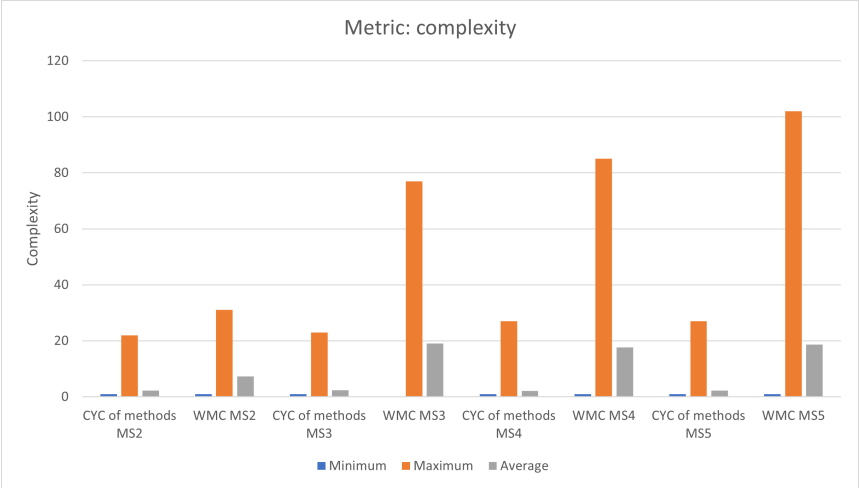

Figure 5: Complexity metric

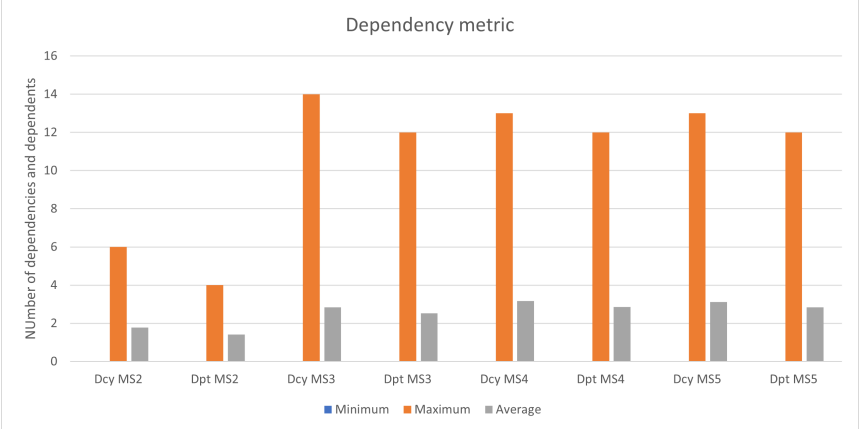

Figure 6: Dependency metric

claim different numbers to be limits to stay within. Some state that 20 is an upper bound for quality, which, as seen in Figure 5, our maximum (for classes) had exceeded since the second milestone already. This is something that we find was hardly avoidable, in particular regarding the movement methods of the cube, which required lots of calculations which were barely compressible. Our maximum CYC of classes is therefore far beyond 20, however, we managed to keep that number for methods fairly low (below 25) and the average for both are beneath 20.

The weighted method count calculates the sum of complexity of methods in a class. Once again, our maximum reaches some high peaks (102 at the end of the project), however, the average stayed below 20, which represents a relatively qualitative project.

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. Both represent the ratio between the number of components present in the project and the number of connections between these components. According to the JArchitect documentation, good Dpts and Dcys are above 1.5. As can be seen in Figure 6, we couldn't manage to keep our minima above that since we have some utility methods which are purely independent, however, our averages stayed above that number for the entirety of the project.