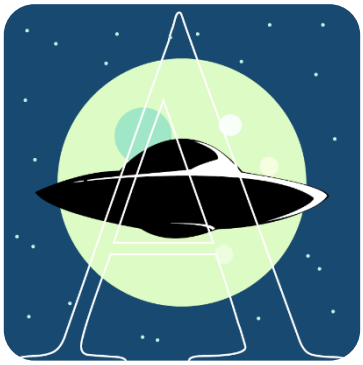
QA-Report

Among Alien 2.0



Contents

1. Introduction

To be able to achieve certain quality aspects in our Game “Amon Alien”, we planned out, managed, and documented our process to the best of our ability. A Software Quality Assurance Report is very important in that it is the proof we must deliver, for a quality Software Product and the red line the group must walk on, to achieve that quality.

2. Goals

From the start, our Primary Goal was to be better and more organized compared to our last attempt the previous year. To achieve this, we used methods of responsible Team Management. Another main Goal to achieve a quality Game was to have clear and responsible leadership. As for the coding aspect, we defined certain rules to have a coherent code structure.

3. QA Concept

To ensure the quality of our game, we needed to have good code readability. So, we used the assignment of descriptive and specific naming, and we will be using unit testing to make sure the game logic is working properly. Considering the naming, we chose descriptive names for classes, methods, and variables, so that their functionality is clear. To enhance the readability of the code we tried to write many smaller functions and not have one big function. This is where we started to check different Code Metrics like Lines of Code per Method, Javadoc Coverage, and Cognitive Complexity. We used these tools to monitor and make sure that our code is easy to understand and easy to debug as we will see later. The following Rules were also implemented at different Milestones to achieve certain quality aspects of our game.

From Milestone 2 we defined:

- New Methods should always have a Javadoc at creation
- New Code should always be communicated to the rest of the team via our Discord Server

Milestone 3:

- We should try and code together as much as possible since the complexity has become bigger with the implementation of the GUI
- The time plan must be held more seriously, and everyone must be responsible for their failures and make them up.

Milestone 4:

- The removal of Magic Numbers and Magic Strings
- The implementation of Logger statements wherever we see fit

4.Methods

4.1. Non-functional Testing

Here we test the Games fundamental issues besides the code aspect:

4.1.1. Compatibility Testing

Compatibility testing involves verifying that the game is fully compatible with the hardware and software it will run on. Which for us meant that the game must be compatible with all major three Operating Systems. There were several times when the game stopped running on Linux or macOS, so we had to analyze and change some aspects of our code implementation.

4.1.2. Usability Testing and Design

This stage of testing was at the very early stage of Milestone 3 when he started discussing our GUI and our design choices. The game GUI shouldn't be complicated but easy to use and intuitive. The goal of this stage was to make sure that the end-user (the players) could in the end play the game fully with no need to use the Manual, and without confusion or frustration due to its user interface or design. That is why we spend a lot of time reworking the GUI design and finally settling on what we believe to be a very user-friendly and fun-looking User Interface.

4.2. Functional Testing

This is where the code testing occurs. And for that we mainly used two methods of Code Testing:

4.2.1. Code Readability Testing

We used various metrics to check our codes' simplicity and readability. The utilization of the IntelliJ plugin Metrics Reloaded helped us tremendously. We checked our Javadoc Coverage to make sure we were at 100%. We checked our Cognitive complexity and our Lines of code per method. That way we were making sure to hold ourselves accountable to the rules we set beforehand to make our code easy to read and understand.

4.2.2. Logging Statements

Event logs record events taking place in the execution of a system to provide an audit trail. That was essential to understand our game and understand where and why our problems occurred. Loggers were heavily recommended in our lecture and by our Tutors, so we used the recommended Logger (Log4J).
Up until Milestone 3, we didn't use the Logger but for a few Logger.FATAL statements. After our talk with the tutor on Milestone 3, we needed to increase the Logger usage. This goal was reached till Milestone 4, and we even implemented different levels of logging. And with Milestone 5 We were so focused on other aspects of our documentation and debugging, that we completely left the logging aspect of our game as is. We used it partly for catching errors

but mostly for debugging. For future projects, the correct incorporation of a logger will be a higher priority from the start.

4.2.3. Unit-Testing

A unit test is a way to test a unit, the smallest code in a system that can logically be isolated. So, we tried to test specific isolated functionalities. We had a plan to test 100% of the Server functionality but we fell short of that because we couldn't implement Mockito correctly. We were able to mock certain aspects of our Game but other aspects we couldn't find a way to mock them correctly. Unfortunately, we had to realize, that we made a mistake during our thought process while writing our tests too. We were told to also write worst-case tests and not only tests that passed because the parameters were passed on correctly (Happy Cases). So, for the next time, we have such a project we will be testing more Sad cases and more detailed cases which go deeper into the functionality of specific methods and functionalities.

5. Results and Discussion

The Results of our used Tools and Metrics will be shown first and then discussed in their specific section:

5.1. Code Coverage: Jacoco

AmongAlien Milestone 3

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch.unibas.dmi.dbis.cs108.AmongAlien.server		0 %		0 %	304	304	982	982	100	100	10	10
ch.unibas.dmi.dbis.cs108.AmongAlien.gui		0 %		0 %	208	208	671	671	121	121	14	14
ch.unibas.dmi.dbis.cs108.AmongAlien.tools		0 %		0 %	64	64	265	265	27	27	3	3
ch.unibas.dmi.dbis.cs108.AmongAlien.client		0 %		0 %	99	99	415	415	32	32	10	10
ch.unibas.dmi.dbis.cs108.AmongAlien		5 %		5 %	12	14	42	46	1	3	0	1
Total	11.122 of 11.133	0 %	724 of 725	0 %	687	689	2.375	2.379	281	283	37	38

AmongAlien Milestone 4

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch.unibas.dmi.dbis.cs108.AmongAlien.server		7 %		4 %	285	308	900	994	84	102	6	10
ch.unibas.dmi.dbis.cs108.AmongAlien.gui		0 %		0 %	200	201	663	664	116	117	12	13
ch.unibas.dmi.dbis.cs108.AmongAlien.client		16 %		7 %	85	98	351	408	21	32	6	10
ch.unibas.dmi.dbis.cs108.AmongAlien.tools		58 %		68 %	28	64	96	266	6	27	1	3
ch.unibas.dmi.dbis.cs108.AmongAlien		14 %		15 %	13	15	41	49	2	4	0	1
Total	9.456 of 11.185	15 %	645 of 722	10 %	611	686	2.051	2.381	229	282	25	37

Created with JaCoCo 0.8.6.202009150832

AmongAlien Milestone 5

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ch.unibas.dmi.dbis.cs108.AmongAlien.server		7 %		4 %	290	314	916	1.011	88	107	6	10
ch.unibas.dmi.dbis.cs108.AmongAlien.gui		0 %		0 %	200	201	668	669	116	117	12	13
ch.unibas.dmi.dbis.cs108.AmongAlien.client		16 %		7 %	89	104	378	438	25	38	6	10
ch.unibas.dmi.dbis.cs108.AmongAlien.tools		60 %		68 %	28	64	98	291	6	27	1	3
ch.unibas.dmi.dbis.cs108.AmongAlien		12 %		13 %	16	18	51	59	3	5	0	1
Total	9.711 of 11.578	16 %	650 of 727	10 %	623	701	2.111	2.468	238	294	25	37

Code Coverage Tools like Jacoco are a percentage measure of the degree to which the source code of a program is executed when tests are run. A program with a high test coverage has more of its code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with a low-test coverage. With that in mind our Game had barely any Tests but the main HelloWorld Test class in Milestone 3. That is why we had almost 0% everywhere.
In Milestone 4 we implemented a few more Tests (12 in total) in which we covered aspects of the game which we were able to test with our lacking implementation of Mockito. For Milestone 5 we didn't change the number of Tests, but the number of code lines changed, so the percentage changed as well. We realize that our Unit-Tests

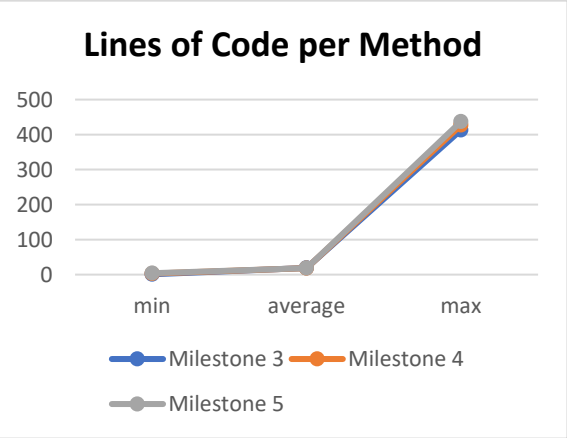
Were very lacking and we will focus on the quality and quantity of our Unit-Tests in our future projects.

5.2. Logging Statements normalized to lines of Code

	Milestone 3	Milestone 4	Milestone 5
Logging Statements	11	124	124
Lines of Code	4360	4526	4585
Ratio	0.0025	0.027	0.027

Because we mentioned earlier what and how we used Log4J and why it won't be repeated here. We can see that our number of Logging Statements drastically increased in number after the feedback from our tutors in Milestone 3. So, we implemented different levels of Logger Statements like INFO. DEBUG and ERROR.
The number of Logging Statements normalized to Lines of Code (Ratio) is still low after our implementation. That is why, in future projects, we will aim to have that ratio bigger than 0.4. This will increase the efficiency the development efficiency of our future Projects. Because of the logger's tremendous ability to help identify errors and problems.

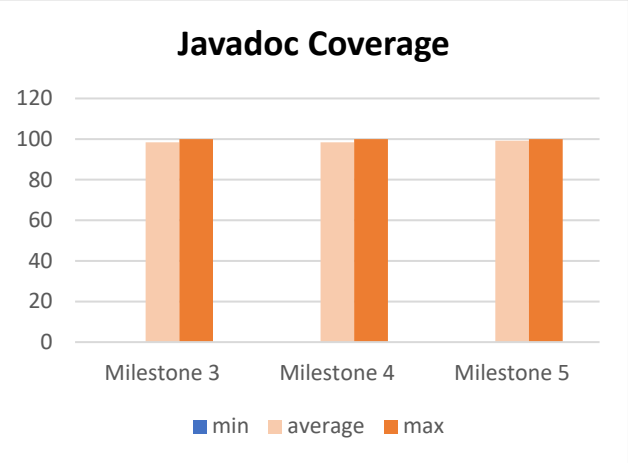
5.3. Three chosen Metrics



Here we tested how much code each method used, and we had varying numbers from 1 to 437 as seen below:

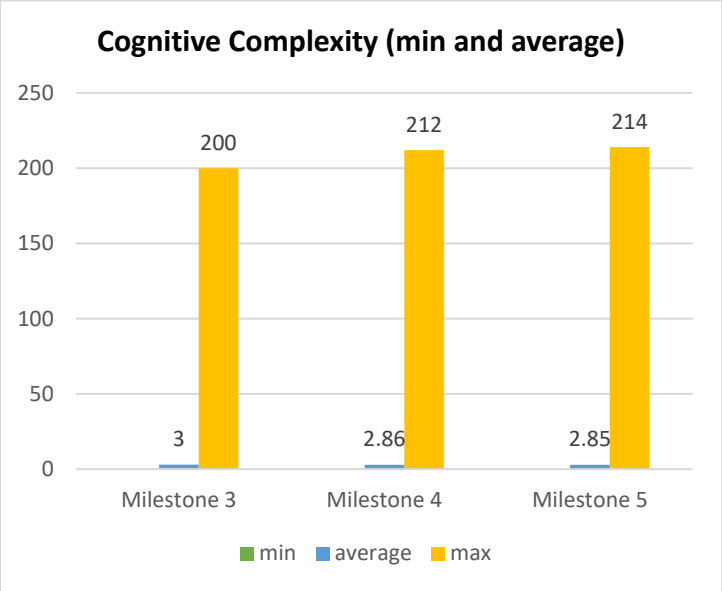
	MS 3	MS 4	MS 5
LoC p. Method	Min 1 Average 19.04 Max 413	Min 3, Average 18.55 Max 427	Min 4 Average 18.7 Max 437

These variations exist because of the Server Protocol Interpret and the Client Protocol Interpret classes, which are very large. Still, the average Method never exceeded 20 lines of Code which were exactly what we aimed for. This makes our code more understandable, and easier to read and debug. What we could improve for future projects is to have even more Methods for some functionalities that we didn't notice.



We were always trying to have a 100% Javadoc coverage but our Metrics Tool Metrics Reloaded gave our Enum always 0% despite having it commented. That is why our min is always 0%. We couldn't achieve a 100% Javadoc coverage in Milestone 3 and 4, because some Methods were overseen. But for the last Milestone, we were able to achieve a 100% Javadoc coverage without counting the Enum.
What we could've done better is achieving 100% Javadoc coverage from the start. Another important improvement is writing

the Javadoc whenever a method gets created. We didn't follow that rule every time and that is why some Methods didn't have a Javadoc in Milestone 3 and 4.



Cognitive Complexity is a measure of how difficult a unit of code is to intuitively understand. It tells you how difficult your code will be to read and understand.
A more complex code is more susceptible to bugs and needs more rework. That's why we tried to have an average of <= 3 in Cognitive Complexity. The lower that Number is the simpler and easier to understand, debug and update our game will be. We were happy with our average of less than 3 during all 3 milestones with our final average being 2.85. The min values were always 0 because we had multiple methods that were very simple and basic in their functionality without using any loops or if statements. The max values are high because of the Server and Client Protocol Interpret class. Lowering that complexity can be achieved by reducing the number of parameters per method and reducing multiple if statements.

6. Conclusion

Our main conclusion from creating a Game and from writing a Quality Assurance Report is that it's always useful to ask for guidance. Without asking our Tutors and other colleagues, we wouldn't have been able to create this game. Even though we had setbacks with logging and unit testing, we are all still very happy with the experience. We learned to comment on methods quickly and clearly. But the biggest lesson we learned from this project and even this Report is Time Management. We will always focus on time management in our projects in the future.