Quality Assurance

Concept

Team culture

In meetings, tasks are assigned to a team member to supervise. This means that that team member will make sure the task will be completed on time, according to specifications and with sufficient quality. This assigning of reasonability should be documented in a Responsibility Assignment Matrix or a list.

Each team member is to ask help from the author when working on their method or class to ensure that nothing will break, or no important lines deleted.

The team is to meet regularly and particularly before a deadline to discuss the plan going forward any current issues and questions.

Each team member is to ask a question repeatedly until they have fully understood the issue. If the other team members can't explain it either, they have probably also not understood the problem and they are to ask someone else or work to understand it.

The team is to regularly check the code, even if they are not working on it, to keep up with any updates and check the others work for quality.

Constructive Quality Management

To prevent an error in the larger network, any new created class or method should be thoroughly tested on its own before being implemented.

Furthermore, the whole code should be tested to make sure that every functionality is working properly as it was in the previous commit. Finally, it is important that all errors must be fixed before pushing a commit to avoid bug fixing while trying to implement new lines of code.

If a bug is found it should if possible be fixed immediately. If that is not possible the person who found the bug should immediately report it to the team with the following information:

- What happened?
- What error / logs did you produce?
- What did you do?
- What version was the bug found in?
- Can the bug be replicated?

Analytical Quality Management

The team is to double check others code whenever working on or with that code to ensure quality, functionality, and completeness.

The team is to regularly sit together to perform an informal review to walk all the team members through important code, particularly after a big update or ahead of a big work effort.

The team is to preform Black-Box testing regularly by disconnecting the server/client, particularly while editing some method or class.

The team is to preform White-Box testing regularly by setting some unusual Input to try to create an error, particularly while debugging or learning another team member's code.

Factor-Criteria-Metrics

To additionally improve the quality of our project we´ll code by considering the Factor-Criteria-Metrics.

- Functionality: object-oriented functionality, using algorithm and data structures
- non-functional factors: clean structure, good documentation, easy to understand and read
- usability: easy to control and use

Javadoc

At the beginning of each class there should be a Javadoc explaining the general functionality of the Class.

At the beginning of each method there should be a Javadoc explaining the general functionality of the method as well as explain how / in what circumstances this method should be called. Within the method itself, sections of the method should be further documented if its complexity warrants it.

Logger

Loggers are a tool to help the programmer output log statements to a variety of output targets. The Logger we are using is Log4j2 (which was introduced in class). The logger can print out 5 different message types that describe their importance. Also, the User won't get irritated by the output he doesn't need or even understand. Furthermore, it is possible to configurate the logger in a way that both the User and the Author of the program get their wished output. In many cases the logger is very helpful to easily locate the error and fix it.

The Metrics

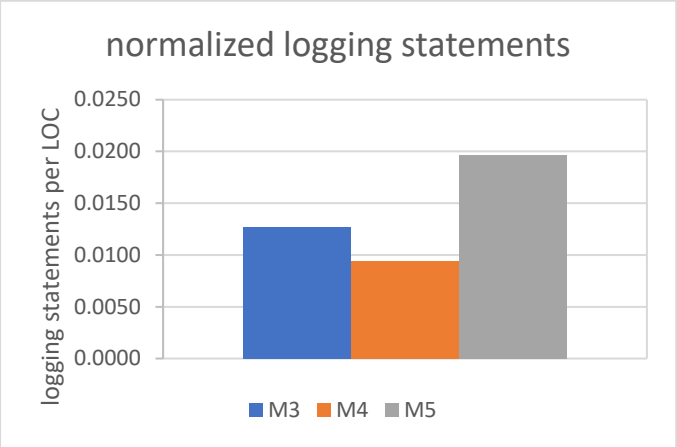
Developers are able to get a better insight into their developing code through code metrics. All sorts of irrelevant comments (such as //,/…/) will be excluded as good as possible. It's also important to keep comments reserved to further explain sections with very difficult logic or domain-specific rules while blocks of confusing and critical behaviour are getting extracted into well named methods to clarify the intend. Meanwhile its significant to limit the comments on short, concise, applicable comments on explaining why it is done and not how. Since long sections of codes are difficult as well as confusing to test and to maintain its helpful to modulate the variety of game aspects to avert infinite classes and the possibility of risking bugs.

- Lines of Code
- Javadoc lines per method
- **Lack of Cohesion:**
  Measure how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tends to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

Results

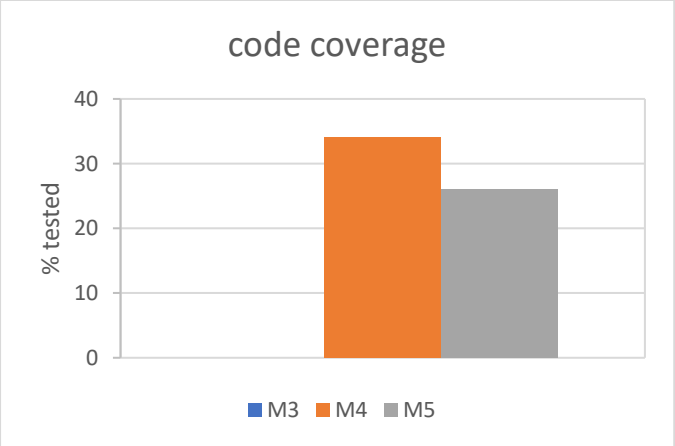
Number of Logging-Statements (normalised to the Lines of Code)

Measured in IntelliJ by searching “logger.” divided by the number of lines of code given by the “statistics” IntelliJ plugin



Code Coverage

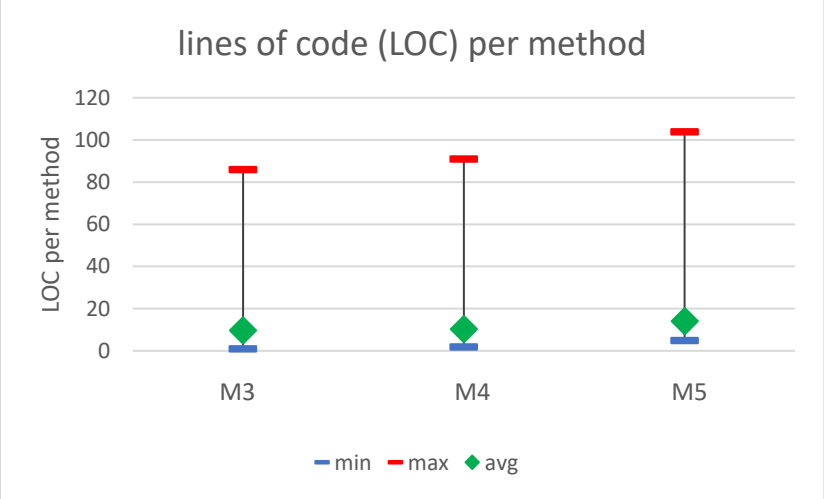
Measured using Jacoco



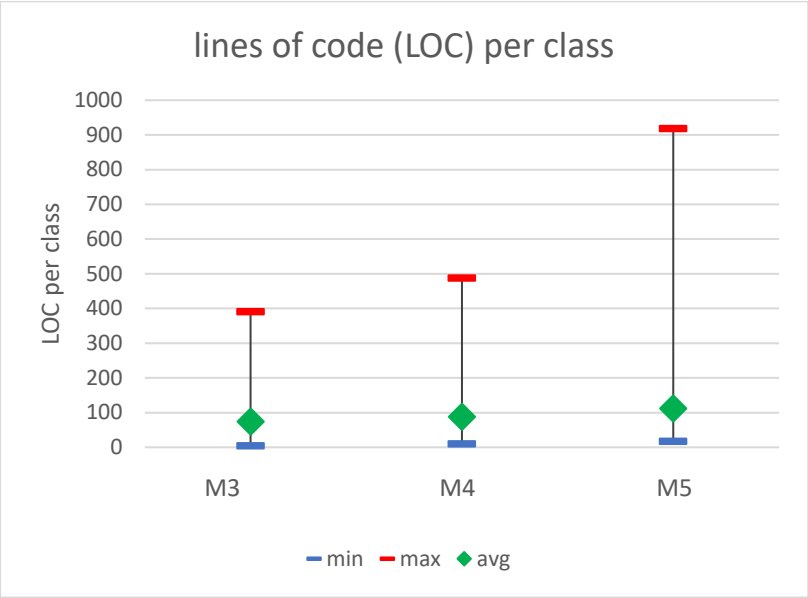
Line of codes

Measured using the “Metrics Reloaded” IntelliJ plugin

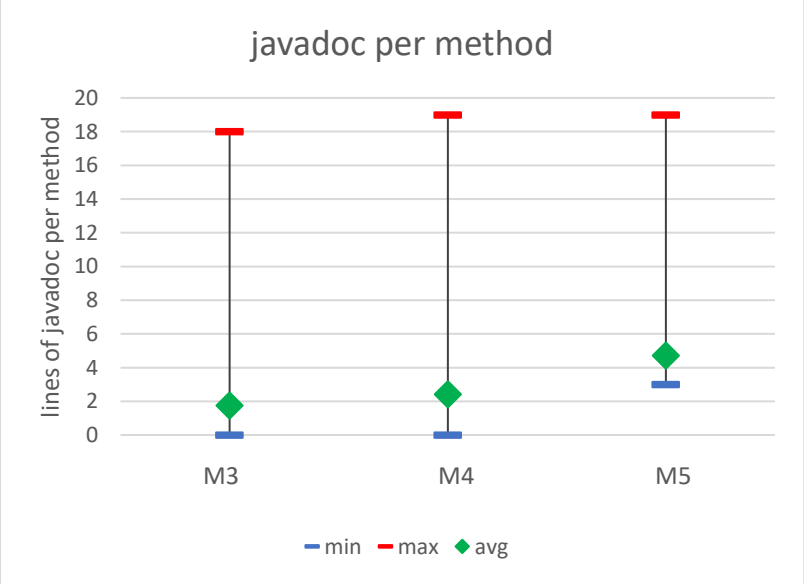
Per method:



Per class:

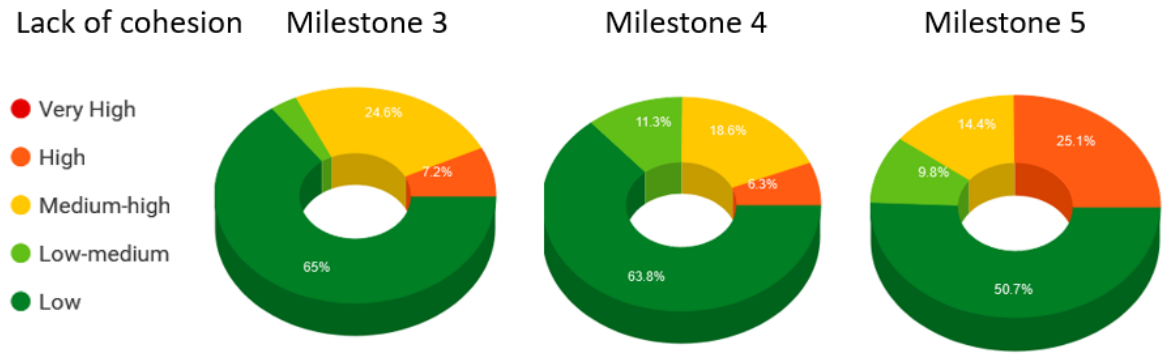


Javadoc lines per method



Class Cohesion`

Measured with “CodeMR” IntelliJ plugin



Results in table form

	M3	M4	M5
lines of code	3070	4577	5344
logging statements	39	43	105
logging statements normalized	0.0127	0.0094	0.0196

code coverage jacoco	0	34	26

lines of code per method			
min	1	2	5
max	86	91	104
avg	9.7	10.4	14.1

lines of code per class			
min	5	10	18
max	391	488	919
avg	74.2	87.7	111.8

javadoc per method			
min	0	0	3
max	18	19	19
avg	1.7	2.4	4.7

low class cohesion			
Very High	0	0	0
High	7.2	6.3	25.1
medium High	24.6	18.6	14.4
Low-medium	3.2	11.3	9.8
low	65	63.8	50.7

Discussion

The logging statements number has increased, although the increase is mainly due to a more thorough use of the logger, instead of the command line output. The number is generally low due to the use command line prints for debugging and deleting once the method worked. However multiple occurrences of deleting and rewriting the same System.out.println were observed by the team, thus a more proper use of the debugging state of the logger combined with the settings of which level should be printed will certainly be used in future projects.

Test code coverage at 34% of the code seemed acceptable, since it is covering the core of the application such as the packet encoding, validating and decoding methods. However, it is known that it would be too low for proper software distribution. Tests were written as an afterthought of the code; thus, the code was written in difficult way to test. Future projects will be written with this in mind, either writing the tests right after the code or using a more test-driven approach starting with tests and then writing the code, which could present a few additional advantages such as being more focused on what must be coded.

Lines of code (LOC) per method is a metric with good results. The averages are reasonably low, indicating good separation of tasks. The maximums do not seem to be too high. The slight upwards trend seems to be appropriate with the increasing complexity of the functionalities of the software.

LOC per class has good results, except for the game map class, which is the maximum of milestone 4. However, it contained functions to print out the map to the console, which is deprecated since the introducing the GUI. It has thus been reduced to under 400 since then. The second outlier is the lobby controller class in milestone 5 approaching a thousand LOC. This class is the result of multiple not fully thought through and hasty additions, approaching the deadline, and would prove a nightmare to debug if adjustments needed to be made. The model view controller design pattern was equally not fully respected resulting in a mix of fxml and direct declaration which doesn't bode well for maintainability. This would of course not be repeated, and more time would be invested to learn the javaFX framework.

The Javadoc lines per method metric has greatly improved from milestone 3, now every method is documented.

Class cohesion was similarly satisfying to other metrics up to milestone 4, due to general good practices employed when coding. The low cohesion class from milestone 3 and 4 is the ClientThread class, which seems to have low class cohesion due its method creating different packets, thus the low cohesion is in coherence with the chosen design. However, the lobby controller, being the second class presenting low coherence in milestone 5, has a lack of cohesion because of the problems mentioned with the line of codes. It isn't a pure controller anymore as it creates view elements and even accomplishes functions that should be in the model.

Lessons learned

The main points these metrics pointed out that needed improvement are:

- Testing, should be done closer to the implementation
- Respecting design rules and patterns until the end