Quality Assurance Report

1 Overview and Execution

1.1 Overview

In the following paragraph the main points of our *Detailed Quality Assurance Concept* as an overview is briefly explained and discussed. In particular, the necessity of each point will be explained with potential goals and afterwards in the section *Results of Kniffeliger's Quality Assurance* the results of some points is discussed in more depth.
The team of *Kniffeliger* agreed on the following methodical and technical methods:

Guidline Document: The software's guidline document created at the beginning of *Kniffeliger*'s development is the cornerstone. The purpose of this central document is not to lose focus on what the team is developing and creating about. As a reference for each developer, it contains the crucial rules governing the game logic, GUI and even the network protocol. This document is only altered if every team member agrees upon to ensure a steady course of the development.
Documentation, JavaDoc: Due to the nature of software development as a team effort, it is crucial that every developer understand everything if necessary. To support this and even encourage understanding and reading code which is not self-written a solid, professional and understandable documentation is mandatory. Therefore every method, field and class should be documented if the name alone is not documentation enough. Moreover, comments in the code are also needed to understand the dynamic of code snippets. This goal will be enforced by using `JavaDoc` and general documentation as commenting or the network protocol.
Coding Conventions: Since uniform code is the best understandable code we agreed on certain coding conventions. This allows with the point before to encourage developers to familarize themselves with not self-written code.
Meaningful Exception Handling: To ensure resilience and robustness of the code of *Kniffeliger* the approach is to address and handle exceptions. Exception Handling should not be viewed as a burden instead more as an opportunity to cope with errors. As a consequence, meaningful exception handling is mandatory.
Logging: Errors and bugs will appear. Due to this unavoidable fact we decided to implement a logging system to log every meaningful code if necessary. Our choice was the `log4j` library. The development should be sped up by using the logger since it enables our developers to debug as fine as they need to and even review older logs by reading the log files to localize errors.
Testing of the Game Logic: The game logic is prone to error because of its natural complexity. To be sure that the game logic does what it is supposed to do Anisja and Riccardo were responsible for writting `UNIT` tests.
Metrics: We decided to use following metrics to quantitavely observe the development:

- **Cyclomatic Complexity** to measure the complexity of *Kiffeliger*'s source code.
- **JavaDoc Coverage** to monitor the overall coverage of documentation.
- **Lines of Code** to point out classes and methods which need a potential slimming. Moreover, the ratio of functional code and commenting code are from relevance.

- **Testing Coverage of the *Game Logic***. To better understand and spot methods which had not potentially been tested we integrated `jacoco` in our project.

Moreover, following organizatorical methods were established:

Shared Allocation of Tasks: A responsibility system via *Google Sheets* was established where each individual was assigned to different tasks and was responsible for keeping other team members informed.

Ticketing System: A bug ticketing system was set up to track bugs. The Goal was to systematically list bugs and assign developers on fixing them.

Regular Team Meetings: Software Development need to be organized thus we scheduled team meetings every week to discuss our progress and localize inconsistencies before arising to problems.

1.2 Execution

Our concept of Quality Assurance is mainly based on self-responsibility. In particular *coding convention* and *documentation* as *JavaDoc* were not enforced by one individual instead every developer had the responsibility to document his own code thoroughly.
For the implementation of the *logger* i.e. `log4j` was Riccardo responsible at first. His task had been to read the documentation and set the logger up. Afterwards he explained the concept of a logger in particular the one we used to the rest of the development team. The team then had the additional task to integrate the logger in the existing project where necessary.

Milestone 2

Task / Status	done	in progress	not started	will do
setting up gitignore	lina			
enforcing utf8 standard	lina			
client chat		lina		
enable changing username			lina	
keep diary up to date		anisja		
login - logout functions	lina			
setting up gradle	riccardo			
parsing of command line parameters for jar		riccardo		
ping and pong				dominique
start game logic			anisja	

Figure 1: Shared Allocation of Tasks

Since organization itself is also an important part for quality assurance we implemented a *Shared Allocation of Tasks* where every developer has access to and is signed in for different tasks. The responsibility of the developer itself is to keep everyone on track about the current status of our tasks fixed in the regular meetings. Therefore, the *Ticketing System* for bugs is an additional signifcant part of our concept.
The responsibility of writing and maintaining tests when code is updated are with Anisja and Riccardo. Both's task is to ensure a fully functional code of the `game logic`.
Moreover, during the whole project metrics for quantitative measuring of quality are calculated. The task is assigned to Riccardo. His job as so-called *Police Officer* is to monitor these metrics and evaluate them. At every meeting of the development team the metrics are compared to older ones and potential problems are pointed out.

2 Results of Kniffeliger's Quality Assurance

2.1 Methodical Quality Assurance

When comparing the results of *Kniffeliger's Quality Assurance* to the expected results one can deduce that they mostly match up. The software's requirements document as centrepiece served its purpose as compact and always available point of information. Since it contained the requirements of the software as well as the rules of *Kniffeliger* with its innovative new *Action Dice Mechanic* it is of enormous value for each developer.
Our *Shared Allocation of Tasks* is besides the document mentioned above an important source of information. Not only information about who is assigned to which tasks are contained but also the progress of each task individually. Hence, every developer always has an image of how the project develops and can even requests a reassingment of tasks if the own's were already completed. Mostly, a reassigning of tasks takes place in our regular meetings. These are always scheduled once a week (see Diary of *Kniffeliger*). In these meetings everyone minutely updates the developing team on the own's progress of tasks. Many times, since some developers were already finished with their tasks, we reassigned them to speed up the developing process. The established coding conventions as the strict rules of documentation are an advantage here. Due to this a developer is capable of easily working on another task.
In these meetings newly reported bugs in the ticketing list are also discussed. Especially, the team then finally decides the responsibility of fixing the bug.

Bug Nr#	Priority (1 high, 2 medium, 3 low)	Bug (location: package/class/method)	Detailed description	Listed by	Responsibility	Status (handled/in progress/will do)
#1	high	GameManager, Dice	NullpointerException when building a DIce array since like with all arrays, the Dice array is first of values null	Anisja	Anisja	FIXED
#2	medium	GameManager, starter/deleteActionDice	when player gets five action dice and plays shift/swap, the next round 4 of the action dice are deleted	Anisja	Anisja	FIXED
#3	medium	ActionDice, steal method	stealing does not work	Anisja	Anisja	FIXED
#4	medium	Networking	can enter lobby even tho we have an ongoing game	Anisja	Lina	FIXED
#5	high	GameManager, startermethod	when player has no action dice then there is a NullPointerException because it cannot read the array length and do the switch case	Anisja	Riccardo/Anisja	FIXED
#6	high	Networking	command is not listed and wrong in documentation	Anisja	Lina	FIXED
#7	high	GUI	broadcast command does not work properly	Anisja	Dominique	FIXED

Figure 2: Bug Ticketing List

This procedure of our meetings mostly ensured the methodical part of the quality of *Kniffeliger*.

2.2 Quantitative Quality Assurance

In this section we mainly discuss our quantitative quality assurance. In particular, the results are discussed over time i.e. each milestone until now. Since *Kniffeliger* is not yet finalized the point of reference will be the 09.05 on the branch `gameGUI`. This branch which is still experimental in some sense (since it is not yet merged with `main`) is the most advanced snapshot of *Kniffeliger*'s finalization.

2.2.1 JavaDoc Coverage

The invaluable value of a `JavaDoc` documentation as documentaiton in general is beyond question. Understanding foreign code is often a difficult and lengthy task. Therefore, a `JavaDoc` documentation as well as a general documentation in the code is a basic requirement. As expected, the documentation allowed to understand the functionality of own as foreign code quickly, reliably and in detail.

As seen in both Figure 3 and Figure 4 we strive to optimize the coverage of `JavaDoc` documentation. Above all, the `gamelogic` stands out. This has a high `JavaDoc` coverage. This is mainly due to the fact that this is the heart of the game that everyone has to understand, but also that two developers are always working concourrently. Anisja and Riccardo therefore always have to ensure that their own code could also be understood by the other person. This is only partially the case in `networking`, as Lina is able to work mostly alone. However, she made sure that the interface, which drives the `game logic` as the `gui`, is very well documented. The `gui`, on the other hand, which is not entered on the diagrams, also has a high documentation density due to the task distribution of Lina and Dominique, just like the game logic.

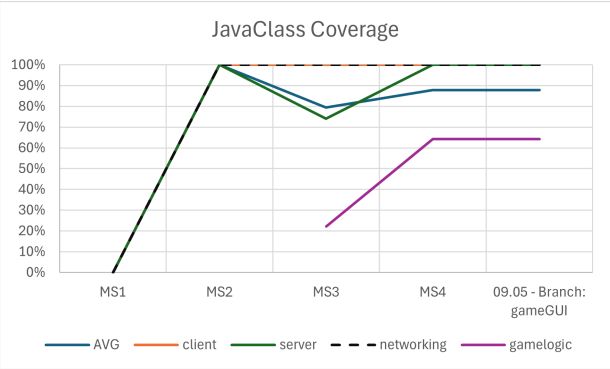

Figure 3: Evolution of the coverage of Classes via `JavaDoc`

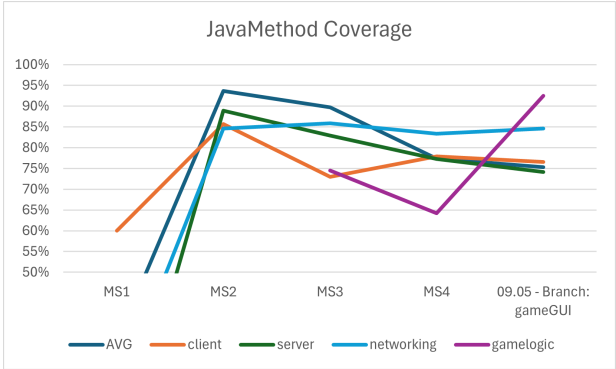

Figure 4: Evolution of the coverage of Methods via `JavaDoc`

The blue trend line **AVG** does not reflects the strive for optimization towards the end of the project but this is mainly due to the fact that a major rework of the `client` and the `server` took place between MS3 and MS4. The top priority always is that interfaces are up to date. In the end, there is this kink to criticize, but in practice our methodical measures such as regular meetings had a dampening and even discussion-promoting effect.

2.2.2 Cyclomatic Complexity

To truly grasp the complexity of a program, we decided, as described in the overview, to use the *Cyclomatic Complexity* metric. As a result, the development team of Kniffeliger not only gained a sense of the complexity or branching of the code, but it also became measurable. The developers aim for a small *Cyclomatic Complexity* figure throughout the entire project. Of course, this is not always preventable, observable in the `gamelogic` or `networking`, which by their nature require more complicated processes with more branches and even exception handling. Still the commandment is the KISS principle. These circumstances are particularly

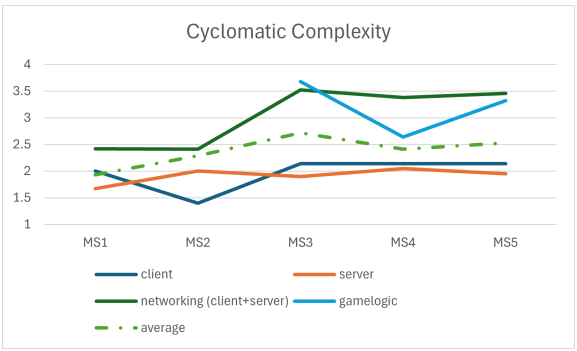

Figure 5: Evolution of the Cyclomatic Complexity throughout *Kniffeliger*'s development.

reflected in Figure 5. It is clear that the developers not only achieved a low figure but also retained it. The `gamelogic` is also particularly noteworthy here. After the first functioning prototype of *Kniffeliger*, a major rework took place, which was supposed to reduce the complexity. The goal here was to reduce the number of possible paths through the code, in addition to more readability, robustness and resilience. The **AVG** trend line in particular reflects this. Finally, the entire development team of *Kniffeliger* is very satisfied with the code and the complexity, while there was a general dissatisfaction at MS3. After the major reworks of the `server`, `clients`, and the `gamelogic`, development became easier again. The good `JavaDoc` documentation played into our hands, which unfolded its full power especially at the interfaces.

2.2.3 Lines of Code

Lines of Code (LOC) alone are not really meaningful, even though they are interesting as a metric, because the more lines of code a software has, the more likely errors will occur. The interesting metric or observation here is the ratio of CLOC (Commenting Lines of Code), JLOC (`JavaDoc` Lines of Code) and actual functional LOC (Lines of Code).

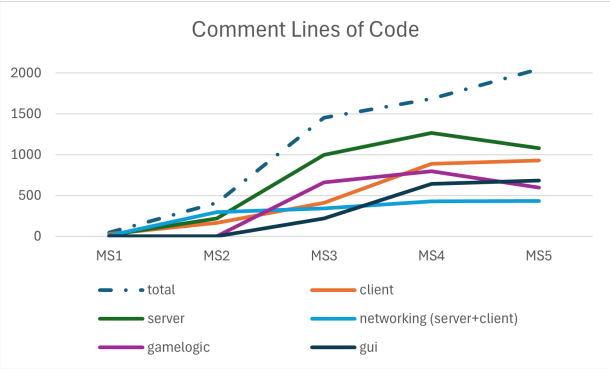

Figure 6: Evolution of the CLOC.

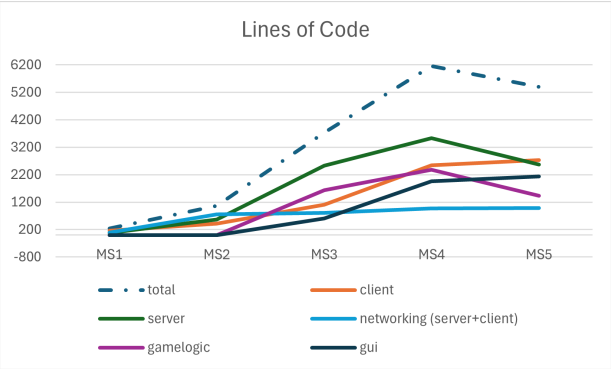

Figure 7: Evolution of the functional LOC

In Figure 6 and Figure 7, one can see that the functional LOC and the CLOC grow in approximately the same ratio i.e. the more functional code is written, the more documentation is provided. This speaks especially for the team's goal to provide a high level of insight into their own code for others. The development team notices this circumstance mainly in the fact that it is easy for individual developers to understand foreign codes. The `JavaDoc` documentation must also be highlighted here in a natural way. The major reworks are also noticeable here again.

2.2.4 Testing of the gamelogic

As the test results in the Figure 8 and Figure 9 below show, the testing process achieves high coverage. However, it is worth noting that the `GameManager` looks untested in comparison. This is a deliberate choice. Understandably, it would be better if high coverage is also achieved here, but this is particularly due to the fact that the majority of the `GameManager` requires network communication. Therefore, it is difficult to test with regular `UNIT` tests. However, there would have been a solution here through suitable technical measures such as the integration of `mockito`, but the development team voted against its use. The consensus is to prefer testing the pathological cases rather than forcing an integration of `mockito`. Summarized, the testing is helpful since it shows the

team locations prone to errors. Moreover, a consistent executing of the written tests assured that the `gamelogic` still worked as it is designed to after altering code.

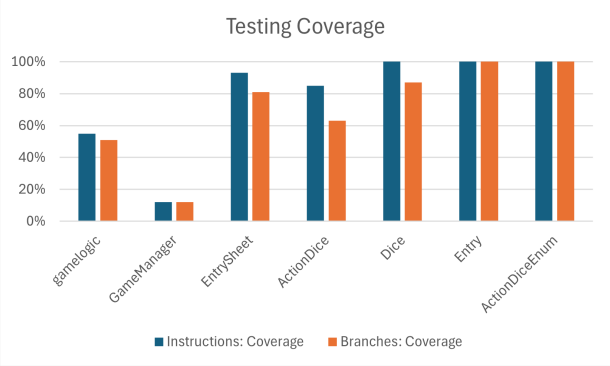

Figure 8: Coverage in testing the `gamelogic`.

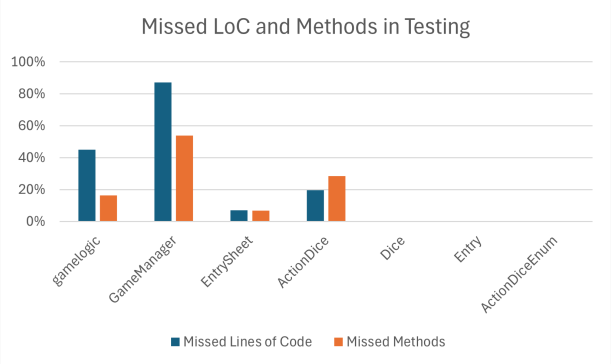

Figure 9: Missed LoC and methods in testing the `gamelogic`.

3 Summary and Insights

Overall, *Kniffeliger*'s development team is satisfied with the quality assurance measures. The methodological measures contribute enormously, as they enable fast, error-free and reliable communication between the developers. The bug ticketing system was a crucial point in our quality assurance. It allows us to systematically record bugs and inconsistencies and immediately refer to the responsible person. During development, the team noticed the very good documentation. In particular, the high quality of the documentation at the interfaces, which often serve as a gateway to a black box for other developers, was enormously conducive to development.
On the other hand, the technical measures were of invaluable value. The integration of a logger allows us to always do live debugging and even look into the past by creating log files. Testing (especially the `gamelogic`) for certain edge cases is also of enormous value to us. The integration of jacoco into our testing process allows us to develop tests explicitly for untested branches in the code. After its integration, through the fine resolution of the data of the coverage of instructions, branches and missing code, tests were specifically developed. Furthermore, the metrics we collected help us to make the code more robust. The Cyclomatic Complexity is an important indicator for the team, because it shows where code can be simplified if possible. The javaDoc coverage as well, because it shows the team bottlenecks in the documentation. Through this, the team noticed that there was still a need for action in some places to increase the density of documentation. Although it must be noted here that there is potential for expansion, but due to the excellent documentation of the interfaces despite the data reality, this hardly weighs.
The quality assurance concept worked out at the beginning unfolds its full effect. The measures we developed were able to fully meet our demands. Especially in this respect, because we have expanded and thus improved our QA concepts through ever further technical and methodical measures. Although there are some areas where the team did not achieve perfect performance nevertheless this did not hinder development. But at the critical points, consistently excellent performance is always achieved. In the area of testing in particular, our developers are satisfied with the technical measures. In summary, the developers perceive our elaborated quality assurance as solid and robust.