

Quality Assurance Concept

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?
- 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.

Each Javadoc should contain the following info in the following order:

- Explanation of the functionality
- Explanation of the parameters
- Author

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

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 responsibility 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.

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 (max 60 characters/ max 200 per method/ max 400 per class)
- Tight Class Cohesion (min: very basic, few lines, more like a help class/average good length and complexity/max too complex difficult to read and understand)
- Javadoc lines per method (0-100: 0 no Javadoc at all, <30 not enough explanation for complicated methods/ >70 unnecessary explanation for an obvious method)

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 configure 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.

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

First Metrics

The first metric: line of codes has been measured using the plugin MetricsReloaded for IntelliJ. The results are shown in the figure below. The Lines per Method were respected with a maximum of 346 (<400 fixed). Lines per method were respected expected for the method `game.packet.packets.Move.decode(Object, String)` with 67 instead of 60.

Method metrics		Class metrics	Interface metrics	Package metrics	Module metrics
class		CLOC	JLOC	▼	LOC
  game.datastructures.GameMap		58	49		346
  game.server.ClientThread		56	47		248
  game.client.Client		17	9		172
  game.datastructures.Robot		34	31		127
  game.gui.LobbyController		14	10		112
  game.packet.AbstractPacket		33	30		110
  game.server.Lobby		15	14		106
  game.gui.StartMenuController		3	0		102
  game.server.ServerSettings		15	10		96
  game.packet.packets.Move		4	4		94
  game.packet.PacketHandler		12	12		79
  game.datastructures.Cell		0	0		69
  game.client.InputStreamThread		8	3		66
  game.packet.packets.Connect		16	10		66
  game.packet.packets.ChatLobby		14	12		64
  game.Test		6	0		63
  game.datastructures.Radar		24	24		63
  game.datastructures.Trap		24	24		63
  game.packet.packets.Update		26	11		63
  game.client.LobbyInClient		4	4		62
  game.packet.packets.Nickname		13	10		60
  game.server.Server		3	0		60
  game.packet.packets.Broadcast		12	12		58
  game.packet.packets.Whisper		5	3		58
  game.packet.packets.LeaveLobby		2	0		57
  game.packet.packets.Chat		12	12		56
  game.packet.packets.JoinLobby		2	0		50
  game.server.PingThread		9	5		50
  game.Main		0	0		44
  game.datastructures.Ore		6	6		43
  game.packet.packets.Awake		13	13		43
  game.packet.packets.Close		11	11		42