Bad quality, decreasing your grade (from more important to less important):

- Missing stuff—diagrams or tables that were asked for are not included.
- Requirements that are ignored or misrepresented in the design
- Errors in the use of UML—a missing arrow head will not cost you points; systematically forgetting cardinalities in a class diagram will.
- Unclear explanations and badly structured text.
- A badly packaged submission, not following the description above.

Not part of the grading criteria:

- Grammatical correctness (as long as it does not degrade the readability of the report).
- Graphic quality of the diagrams (as long as they are clearly readable).
- Layout of the documents.

In the unfortunate case that your group has to resubmit an improved version, you will receive a list with specific requests that have to be satisfied in order to get at least 5.5 as final mark when the project is resubmitted in Week 10.

Programming project

During the module, you have learned all the tools you need to build a high quality software product. In this project, you will be putting this knowledge in practice by developing your own software product. Together with your partner, you will be developing a client-server multiplayer game. The game description will be available on Canvas when the project starts. The goal is to demonstrate the design and programming skills you acquired.

The game you will be developing will be based on a client-server architecture. This means that you essentially have to deliver two products: a client and a server. The server controls the game, and the client serves as one player of the game. Of course, the client and the server will need to be able to communicate with each other. Since it would be much more fun if your client and server could not only communicate with each other, but also with the client and server of your fellow students, we have already designed a communication protocol for you to use. You can find the protocol on Canvas when the project starts. In order to verify whether your game adheres to this protocol, a pre-compiled reference client and server is available on Canvas by then as well.

Since your client and server should be able to communicate with all other clients and servers developed by your fellow students, there is an opportunity for competition. You should not only be able to play the game as a human, your client should also be able to play by itself! This means your client needs to have a computer player. At the end of this module, there will be a tournament to determine who has the best computer player. As an added incentive, the winner and runner-up per house will receive bonus points on their final grade.

In order to achieve this goal of having a fully functional client-server game implementation, we expect you to adhere to all coding standards and good practices you have learned throughout this module. We expect you to deliver a preliminary design at the start of the project and a complete (design) report at the end. We expect your code to be structured in a sensible way, by distinguishing different responsibilities and by designing and implementing the software such that individual parts like the user interface could be changed without affecting the rest of the code. We expect good documentation and good application of the object-oriented programming concepts that you have seen throughout this module. Finally, we expect you to have extensively tested your product according to a clear plan in your report. Testing is a very important part of your grade. You are expected to write automated unit tests for important parts of your project, especially to ensure that your implementation of the game logic is correct. You are also expected to write system tests: instructions for a developer to test certain aspects of your software.

The next sections will go more into the details of the requirements and the grading criteria. Detailed grading criteria can be found in appendix C (page 133 onwards).

Detailed description of the client and server functionality

Client-server architecture As mentioned above, you will build a separate client and server. Two clients can connect to the same server, after which a game can be initiated. As the game is played, updates (e.g. executed moves) are communicated between client and server via the predefined protocol. Using this information, both clients keep track of the state of the game, and can prompt the user for possible moves via a user interface. Both client and server should verify the legality of a move before sending it over the network. When a game is finished, the users should be able to play a next game immediately after: client and server should not need to re-establish a connection.

Server functionality Since many players can join the server, it also needs to be able to host multiple games simultaneously. On top of that, it could be possible that multiple distinct servers run on the same computer. For this reason, the server should prompt the user for a port number when it is started. When the port is already taken, it should ask the user for an alternative port to try again. Once initialized, no interaction between user and server should be required to play games.

Client functionality In order to connect to the server, the client should request the server IP and port from the user when starting. Additionally, the user should enter a "username", which is used for various purposes in the protocol. This name should be unique in the server: if the name is already taken, the client should prompt the user for an alternative. Once connected, the user should wait for an an opponent and then start playing. Ideally, the TUI also helps the user where possible, for example by having a "help" command to see a list of available commands. When it is the player's turn, the user should be able to enter a move or request a (legal) move as a *hint* from the client. Your client TUI should be designed with novice players in mind, for example your family members or friends should be able to play a game using your client TUI.

Computer players You are required to integrate an AI into your client, which can play the game instead of a regular human behind a keyboard. This AI should have an adjustable difficulty level. The user should be able to indicate who should decide on moves (AI and corresponding difficulty level, or a human) via the TUI at the start of the game.

Expecting the unexpected Unfortunately, you cannot expect all parties to behave as expected. There might be clients and servers that do not adhere to the protocol. This might be because of an implementation mistake, or a malicious attempt by a client to cheat. When playing a game using your client on a server not adhering to the protocol (or vice versa), your product should not crash under any circumstance. This means that your product (both client and server) should be able to handle randomly dropping connections, malformed networking messages or illegal moves without crashing themselves. When a connection to a server drops, your client should inform the user what happened and then terminate cleanly (e.g. without some thread dumping a stacktrace or hanging forever). When a connection to a client drops, your server should inform the other client (when in a game) and continue operating normally. It is *never* allowed for your software to crash or hang due to getting unexpected (network or user) input. This is also related to the concept of *defensive programming*: eliminating as many foreseeable problems as possible so that the software remains functional under unforeseen circumstances.

Description of other code requirements

Design and quality of code The stability and the maintainability of your software is heavily dependent on the design of your program, both the high-level architectural design into components and packages, as well as the low-level detailed design using design patterns and the principles of object-oriented programming. Your code should make use of design patterns that we teach in the module, in particular you should make use of Listener patterns as well as the Model-View-Controller pattern such that someone could write a new user interface without modifying the rest of your program. Your code should follow the practices that we have seen during the module: proper use of inheritance, encapsulation, abstraction and composition, as well as following Java code conventions (as enforced by Checkstyle). In addition, you should make use of (custom) exceptions where appropriate.

Tests In order to guarantee smooth running of the software, as well as prevent bugs in the future, proper testing should be done. You are expected to write automated (unit/integration) tests, covering as much of your codebase as possible, but at least the implementation of game logic. Start with writing tests for the game logic, and continue adding tests for other parts of your system afterwards. Furthermore, you should keep good testing practices in mind: make sure each test only tests one clearly defined piece of code or one clearly defined case. In addition, we expect you to write system tests in the report (see below).

Documentation Like any piece of software, proper documentation should be written to aid future readers in understanding the code. For Java, this comes down to having proper Javadoc for classes and methods, but also helpful in-line comments to explain *why* something is done the way it is[2]. For the classes in the game logic, you should also add proper pre- and postconditions to the Javadoc documentation. In practice, you should write Javadoc before implementing classes and methods and you should write comments while writing code. Writing documentation helps to think before you code.

Description of the report contents

The source code is not the only deliverable for the project. You are also asked to submit a report, in which you explain the architecture of your software. Additionally, you are asked to elaborate and reflect on various decisions that were made before and during the implementation, as well as the testing methodology. The paragraphs below explain what sections your report should consist of, and how they relate to each other. For the sections related to the design of your code, your target audience is a future software maintainer. Focus on what would be important to someone who might need to adapt your code in the future.

Justification of minimal requirements In the grading section below, you can find a list of minimal requirements. Not adhering to these will *strongly* reduce your functionality grade. However, they can be implemented in various ways and in various places. In this section of the report, you briefly explain exactly *where* and *how* you comply with these requirements, for example by listing when it can be triggered and using which command in the TUI, and by indicating in which source file to look. Go over all requirements in a systematic way, for example by listing all requirements, and why you adhere to them, one by one.

Explanation of realised design In this section, you explain the design that you realised in the end. It is your objective to guide the software maintainer through the classes and packages that your project is composed of, and (where necessary) to explain what a class does and how it relates (i.e. what purpose it serves) to other classes. To give this explanation some structure, we expect you to create a list of responsibilities (e.g. "manage the game board") and a list of classes and the relationship between the responsibilities and the classes and to explain your design choices. You should also justify how you divide your program into different components and packages. You should use class diagrams to aid your explanations[3] where appropriate. In addition, you should use sequence diagrams to show how classes cooperate during certain execution flows, such as processing a particular user or network input. Feel free to pick other execution flows, however: whatever is most appropriate for your project. After all, your goal is to explain all major design decisions to a future maintainer of the code.

Concurrency mechanism Both your client and server will make use of multiple threads to function properly. As a consequence, concurrency issues could pop up in multiple areas in both applications. In this section of the report, you should explain where new threads are created and what purpose they serve. You should also identify all variables that could be accessed by multiple threads, what could potentially go wrong, and what you did to ensure that no problematic race conditions occur. Since a simple concurrency design is often the best, this section does not have to be particularly long, but it should answer all questions about concurrency in your application. It could be that you discover a flaw in the design, which is discovered so late that it cannot be resolved before the deadline. In that case, you should explain what the issue is, and how you would have solved it, if you had more time.

[2]What is explicitly *not* requested, are comments explaining *what* is being done. Your function/variable naming should already make that clear. Reasons for doing something are much more helpful, especially when those reasons are not immediately obvious.

[3]These class diagrams only need to display a portion of all classes. As an example, you can have a class diagram with all model classes before the actual explanation, so that the explanation can go over the diagram in a clear way. It is fine if the same class appears in multiple diagrams.

Reflection on design As opposed to the "explanation of realised design", this section should contain things that you did *not* end up doing. Compare the realised design with the design you submitted at the start of the project. In what way did things improve? Why were these changes necessary? What part of the initial design worked out well? You should also look ahead: if you were to make this project again, what part of the design would you do differently? Why could you not make such improvements during the project?

System tests Many tests cannot easily be fully automated. One category of tests for which this is particularly true is system testing. This section should systematically describe the procedures to test the system as a whole (client and server either individually or cooperatively). The tests should be reproducible by a future maintainer with no pre-existing knowledge about the software. You should also list the expected outputs for the different steps. Focus on visiting all features of the client and server, including the "bad flows", such as disconnections and unexpected inputs. See also appendix A (page 119 onwards) for more information on system testing.

Overall testing strategy The system tests above are only one part of your testing strategy. There should be a strategy involved in testing the different parts of your system using your unit, integration and system tests. In this section, it is your goal to convince the maintainer that your testing strategy has led to a stable application. The recommended way to do so is by combining the results from code metrics (e.g. test coverage or class complexity) with the contents of your tests in a concise manner. Explain why the tests are structured the way they are, and why this leads to a stable product. Explain how one test covers what another test does not, why certain features are tested the way they are, and why there is a focus on a particular area. Undoubtedly, you will have tested some part of the system less than others. Explain what this component is, and why (despite that) your testing strategy is still a solid one.

Reflection on process This is, quite likely, the first time that you have made a software project of this size. As such, it is only natural that not all things went as well as predicted. Perhaps some activities should have been done earlier, before or in parallel with some other activities, or perhaps there was an issue in the collaboration process somewhere. In this section, you should describe the positives and negatives of your initial planning and the collaboration. You should mention how to improve on the things that did not go well, as to prevent similar issues in future projects. This complete section should be done *individually*. Your report should clearly indicate which group member produced which reflection.

Possible Extensions of the Application

If you have sufficient time, you can extend your game according to the suggestions below. Implementing more extensions makes your project more advanced and will result in a better grade. For each extension, you can get a maximum number of points. *Extensions will only be graded if the application without extensions is sufficient for a pass, i.e., if it is graded with at least a 5.5.*

Some of the extensions may require additions to your networking code, as described in the protocol documentation for these extensions. You should make sure your client and server still correctly work with the servers and clients from different pairs, after the extended functionality is implemented.

Chatbox (max +0.3 points)

A game application implemented as described above can only be used to play the game. It would be nice if we had the possibility of communicating with your opponents during the game. Therefore, a possible extension is to extend the server and client UI to allow players to send short pieces of text to the a global chat or to individual players.

Ranking (max +0.3 points)

The server could maintain a player ranking, acquired by maintaining statistics about the performance of players on the server. Possible statistics are winrate, total amount of wins or more advanced metrics such ELO-rating. It is acceptable if the calculation is done using only data about the period that the server is currently running: persisting the history to storage to "survive" reboots is not required.

Authentication (max +0.4 points)

Since a client can freely choose its username, a client could pretend to be someone else (for example, to influence the ranking). One way to prevent this is through authentication. The protocol describes a way to authenticate a player using public-/private-key algorithms. This key should be stored locally, to ensure that the same key-pair can be used for subsequent connections from the client. During connection initialization, the server verifies the authenticity of the client using a challenge.

Encryption (max +0.4 points)

While in principle nothing secretive is communicated, it might be desirable to add encryption to your client and server to ensure message confidentiality. The protocol describes extra commands to allow for shared key negotiation, which prevents eavesdroppers from listening in on your client-server communication.

Packaging for Submission

The implementation must be handed in as a *single ZIP archive* file via Canvas. This file must include your *report as a single PDF file* and a *ZIP archive of your implementation code*.

You can generate your implementation archive in INTELLIJ using the EXPORT... item from the FILE menu, and choosing PROJECT TO ZIP FILE. The ZIP archive with your implementation should contain the following:

- A directory `src` containing all source files of the self-defined classes, stored in a single directory hierarchy.
- (Optional) A directory `test` containing all source files of the unit test classes, if you use a separate test sources root.
- A directory `docs`, containing the documentation of all self-defined classes (HTML pages generated with JavaDoc) in a directory hierarchy that is separated from the source files.
- Any non-standard predefined classes and libraries, to be included as `jar`-files.
- A `README` file, located in the *root folder* of the ZIP archive, containing:
  - Information about building and testing your software, indicating, for example, which directories and files are necessary and which conditions apply, such as versions of libraries.
  - Steps on how to start the game, including example start commands if applicable.
  - See also `https://www.makeareadme.com` for suggestions on writing a great README.

Before submitting this package, make sure your program compiles without problems with the files that you will submit! Check your submission before submitting!

Warning: Typical issues that make the installation and compilation procedure fail are names and paths or hardcoded URLs. *Test this before submitting your project!* If your program contains references to the file system, e.g., to load image files, make sure these references are platform-independent. It is possible that some user will run your application under a different operating system than the one you used to develop the application. For further information, see the documentation for `java.lang.File` and in particular the constants `pathSeparator` and `separator` defined in this class. You could also test your application in a freshly installed virtual machine with an operating systems that differs from your own to be absolutely sure about the compatibility.

Grading

The project is graded according to the rubric that can be found in appendix C (page 133 onwards).

Rubrics The rubrics consist of five columns, where each column corresponds to a specific grade (indicated at the top of the rubric). For each row, the cells indicate what is required to get the corresponding points on that component of the grade. If multiple cells are applicable, the grade corresponding to the *leftmost* applicable cell is selected.

Desk reject If a deliverable is missing (source code or report) or required parts (game logic, server/client) are clearly unfinished, then the project is rejected and not graded. The server and client are deemed "not clearly unfinished" if it is possible to use the server with the reference client and the client with the reference server to play a game as a human player and with the AI.

Initial design You need to hand in an initial design in week 8. This assignment is mandatory. The purpose of this design is to get you to think about the design of your project in advance. This initial design will also be used to reflect upon in the report. If you do not hand in an initial design, you automatically fail that part of the report. What exactly you hand in as an initial design and how much you want to design, is up to you. The amount should be sufficient to reflect back upon in your final report. It could for example consist of a class diagram and/or sequence diagrams, a list of responsibilities and a preliminary list of classes and components/packages, perhaps already some descriptions of what threads you are going to need, some description of where you expect to need synchronization/locking to prevent concurrency issues, et cetera. You do not get feedback from teachers or teaching assistants on the initial design; it is meant for reflection in the report. However, it is probably a good idea to submit a design of reasonable quality, not only to improve the quality of your end product but also to be able to do a proper reflection. If you miss the deadline for this, you will simply get 0 points for "reflection on design" in the report.

Midway submission You can submit a partial project in week 9. If you do this before the deadline and the following points are present with reasonable quality, then you get +0.5 bonus points if your project is sufficient for a pass without the bonus points.

- All game rules are implemented.
- All public classes and methods of the game logic have Javadoc.
- Complex methods of the game logic have comments.
- There are unit tests for the game logic, including at least testing whether a move is performed correctly, testing a gameover condition, and testing a random play of a full game from start to finish including checking the gameover condition.
- There is an initial version of the report including a section on the overall testing strategy with incorporation of coverage metrics of the game logic and an explanation of the test plan for the game logic.

During grading, the grader will check presence of the above items in the midway submission. If present and the entire project is sufficient, then the bonus point is granted.

Functional Requirements The following requirements are deemed "crucial":

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively.
2. The client can play as a human player, controlled by the user.
3. The client can play as a computer player, controlled by AI.

If the project is clearly not sufficient to fulfill the above requirements, then it is not graded. In addition, the following requirements are deemed "important" and need to be addressed in your report ("justification of minimal requirements")

1. When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.
2. When the client is started, it should ask the user for the IP-address and port number of the server to connect to.
3. When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.
4. The client can play a full game automatically as the AI without intervention by the user.
5. The AI difficulty can be adjusted by the user via the TUI.
6. Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between.

7. All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.
8. Whenever a client loses connection to a server, the client should gracefully terminate.
9. Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.

These important requirements are part of the functionality grade in the rubric. We suggest that you also write system tests in the report to test these requirements.

Grade calculation The grade G will be between 1.0 and 10.0, and calculated using the formula $G = \min(10, \max(1, 0.2*F + 0.4*S + 0.4*R + B))$, where:

- F is the functionality grade, calculated as a weighted average using the percentages from the functionality rubric,
- S is the software grade, which is calculated in an identical manner using the software rubric,
- R is the report grade, which is calculated in an identical manner using the report rubric, and
- B is the sum of the accumulated bonus points or 0 if $0.2*F + 0.4*S + 0.4*R < 5.5$.

Activities and important dates

Tournament In Week 10, a tournament will be organised in which the different computer players will compete against each other. Bonus points can be gained by scoring high in the tournament within your house. A full point is earned by the winner, 0.5 points is earned by the runner-up. The bonus is added to the grade in a similar way as extensions, i.e. as a flat bonus to the overall grade provided that the grade is passing without the bonus (part of B in the grading formula).

Important Dates The following table lists the several important dates for the project.

Week	Day	Hour	Activity
Week 6	Tue	8–9	Kick-off lecture on the programming project
Week 7			Discuss project planning with student assistant
Week 8	Wed	23:59 CET	Initial design deadline (via Canvas)
Week 9	Wed	23:59 CET	Midway submission deadline (via Canvas)
Week 10	Wed		Tournament (schedule announced separately)
Week 10	Wed	23:59 CET	Submission deadline

Suggestions We have a few tips to get you started.

- Read the entire project description and the rubric first
- Use the rubric as a checklist before submitting
- Write Javadoc before you implement classes and methods
- Write comments while coding, not afterwards
- Use Checkstyle from the start
- Programming is a team effort. Use the group contract to make an agreement with your partner on expectations and how to do the project. Meet daily to discuss progress.
- While pair programming is not mandatory for the project, we recommend that you use it.