Implementation project description

1 Introduction

In this project you will design and implement a networked multiplayer board game. The main goal of the assignment is to gain hands-on experience building well-structured, maintainable, and testable software. The detailed game description will be published at the start of the project.

The software you will develop follows a client-server architecture. The server manages the games and enforces the game rules, while each client represents a single player. At least two clients must connect to a server in order to play a match. In addition to allowing human players to play games on the server, you will develop an AI client that can autonomously select moves and play complete games. Altogether, you are asked to develop **three programs**: the **server**, the **human client**, and the **AI client**.

The client and server communicate according to a predefined **communication protocol**. By adhering to this protocol, your implementation will be compatible with the clients and servers of other student teams. At the end of this project, there will be a **tournament** to determine which team created the best AI.

To support development and testing, a pre-compiled **reference client** and **reference server** will be available. These should be used to verify that your implementation follows the protocol correctly. During the functionality sign-off, your client and server will be tested against the reference implementations to assess whether your project meets the functional requirements.

2 Sign-off points

To pass the project, you need to pass the following **mandatory sign-off moments**:

1. Initial software design (Week 8): You submit your initial design, including a list of classes and four diagrams, and discuss it during the sign-off session to receive feedback. This design will form the foundation of your implementation.
2. Functionality (Week 10): Your software is tested against the reference implementations to verify that it meets the minimum functionality requirements.
3. Code quality and testing (Week 10): Your implementation and tests are **graded** (rubric) in a joint session, with each team member individually asked to explain parts of the code. You will receive feedback on code structure and test quality.
4. Reflection (Week 10): You prepare slides with talking points and participate in a reflection conversation about your design and development process.
5. Tournament (Week 10): You compete in the AI tournament with the other teams.
6. Skills assignments on group collaboration and planning. (Not described in this document.)

For an overview of the project weeks, see Table 1. The initial software design, functionality, and reflection sign-offs each have a required submission on Canvas, which must be completed before the corresponding session. The initial software design must be submitted by Wednesday of Week 8, 13:44. The code must be submitted by Monday of Week 10, 23:59. The slides for the reflection sign-off must be submitted by Tuesday of Week 10, 23:59. In addition, there is an optional feedback session in Week 9 where you can present your progress and receive feedback.

If you fail to attend or do not meet the minimum requirements for a sign-off, you cannot pass the programming project. Exceptions may only be made in documented cases of illness or other approved circumstances.

		Monday	Tuesday	Wednesday	Thursday	Friday
8	1–2	Wrap up practicals & Start on Project				
	3–4			(Submit by 13:44)		
	6–7		TAs available	Initial Design Sign-off	(Backup)	Calculus Exam
	8–9					
9	1–2			Skills Resit		
	3–4					
	6–7	TAs available	TAs available	Midway feedback	TAs available	
	8–9					
10	1–2		Functionality Sign-off	Reflection Sign-off		(Backup)
	3–4					
	6–7	TAs available	Code & Testing Sign-off		Programming Exam	Tournament & End of Module Drinks
	8–9	(Submit by 23:59)	(Submit by 23:59)			

Table 1: Schedule of the project weeks.

Special cases Students that don't require the Design course, specifically double-degree AM-TCS students, don't need to submit the four diagrams, but they still need to prepare a serious and credible design that answers the questions in Initial software design. Retake students that require a 1-10 grade receive a grade $5.0 + 0.5 \times SQG$, where SQG is the software quality grade from the Code quality and testing signoff.

3 Functional requirements

This section describes the functional requirements of the project and should be carefully studied before proceeding with the initial software design and the implementation.

Server functionality When the server starts, it must prompt the user to enter a port number on which it will accept connections. If the chosen port is unavailable, the server should request an alternative until an available port is provided. After startup, the server must run autonomously without requiring further user interaction. The server must:

- Accept connections from multiple clients concurrently.
- Allow multiple users to queue for and play games simultaneously.
- Match clients who are waiting for a game into pairs and start games automatically.
- Enforce the rules of the game and maintain authoritative game state.
- Broadcast valid moves to all players in a game.
- Support uninterrupted gameplay: when a game ends, clients should immediately be able to queue for a new game using the existing connection.
- Handle client disconnections by ending the affected game, informing the remaining player, and continuing to operate normally.

The system must allow human and AI players to participate seamlessly. The server should not make any distinction between human and AI clients; it simply manages games based on the communication protocol.

Client functionality The human client must provide a text-based user interface (TUI) designed to be clear and accessible for novice users. A user with no technical background should be able to play the game, such as your friends and family members. When the client starts, it must prompt the user for the server's IP address and TCP port number. Once connected, the client must:

- Prompt the user for a username and retry if the username is already in use.
- Allow the user to queue for a game.
- Inform the user when a new game has started.
- Clearly display the game state and user prompts.
- Allow the user to enter moves in an intuitive format.
- Provide assistance through features such as a help command.
- Either provide a visual indication of valid moves, or allow the user to request a hint (a valid move).

The client must not display protocol messages or require users to manually construct or interpret network messages. All protocol handling must be internal. After a game finishes, the client must allow the user to queue for a new game immediately without having to restart the client or reconnect to the server.

AI functionality Similar to the human client, your AI client should first ask for a server's IP address and TCP port number, as well as a username. If the username is already taken, the AI client should ask again. After successfully logging in on a server, the AI client should either immediately queue for and play games without requiring further user interaction, or allow the user to configure the AI (for example select the difficulty) and give an explicit command to queue for and play a game. The AI should be able to play a full game without further interaction. The AI client could support at two or more difficulty levels, with higher difficulty incorporating deeper or more time-consuming decision-making. For example, instead of choosing a random move, your AI could try every move and decide which move results in the best next board state. Ideally, the AI should support configurable thinking time, allowing the user to specify, for example, the number of milliseconds to consider each move.

Network communication All communication between clients and servers must use the predefined communication protocol. This protocol must be followed **exactly** and may not be modified. Correct implementation is essential for interoperability with the reference server and client. All handshake procedures, feature negotiation, and gameplay communication must adhere precisely to the specification. The client and server must both track and validate game state to ensure that only legal moves are transmitted. Illegal moves or malformed protocol messages may never be sent.

Ideally, protocol handling (parsing and generating messages) should be implemented as an independent component, separate from the user interface or game logic, to support maintainability and a clean software design. Your implementation should handle unexpected protocol messages, malformed networking input, and loss of connection gracefully. No component of the system may crash, hang, or produce unhandled exceptions due to network or protocol errors.

Expecting the unexpected The components of your software must be robust. The system must remain stable under unexpected conditions, such as invalid user input, malformed or unexpected protocol messages, illegal moves from another client and network interruptions or client/server disconnections.

In all such cases, your software must handle the situation gracefully. It is never permitted for the server, human client, or AI client to crash, hang, or enter an inconsistent state. Instead, they should report the issue to the user if applicable, terminate the affected game or connection cleanly, and remain operational where possible. For example, if the connection with the server ends, the client should not crash, but it should gracefully exit or restart and display a clear message to the user.

This aligns with the principle of **defensive programming**: anticipating and safely handling invalid or malicious behaviour to ensure reliability. Typical things to consider here include correctly handling checked exceptions, but also handling or avoiding unchecked exceptions such as a `NumberFormatException`, an `ArrayIndexOutOfBoundsException`, a `NullPointerException` and so forth.

Possible extensions of the application

After implementing the required functionality, you may choose to add optional extensions. These are intended for students who want to explore additional features, apply more advanced concepts, or just like a challenge. Any extension must preserve compatibility with the reference client and server, and correctly work with the servers and clients from other pairs. Most extensions require additions to the protocol that are described in the protocol description.

Named queues Instead of joining a queue to find a random opponent, players can choose a named queue to play against a specific opponent. When two players enter a queue with the same name, the server should create a game for those players. The `QUEUE` command in the protocol is modified for this extension.

Chat functionality Add support for text communication between players. Messages may be sent globally to all connected users or directly to a specific user. The user interface should allow viewing and sending messages without interfering with gameplay. The protocol includes commands for this extension.

Ranking The server may track player statistics such as number of wins, win rate, or an ELO-style rating. Rankings only need to be maintained during the current runtime; persistent storage across restarts is not required, but may be implemented if desired. Specific protocol commands are added to obtain the scores of players in the server.

Authentication and security Implement player authentication using public-private key pairs as described in the protocol. The server verifies the players identity during the handshake. Encryption of messages between the client and the server is built into the protocol extension.

Color and sound Enhance your client using ANSI escape codes to add color to your text-based interface, and simple sound effects using the Java Sound API.

Graphical user interface Implement a graphical user interface (GUI), for example using Swing, in addition to the text-based interface. The GUI should present the same functionality as the TUI and interact with the rest of the code through the same interfaces.

4 Software quality requirements

Developing a fully functional networked game involves more than just making the system work. Your code should reflect the software engineering principles covered in this module to ensure it is well-structured, maintainable, and testable.

Design and quality of code Your program should be organised into clearly defined components (packages and classes), each with its own well-bounded responsibility. A good architecture separates concerns such as user interaction, game logic, and network communication, such that changes to one part of the system do not require modifying others. Low coupling and high cohesion are key goals. Object-oriented principles taught in this module, including encapsulation, abstraction, composition, and, where appropriate, inheritance, must be applied consistently to achieve such separation. In particular, each class should have a clear purpose and follow the single responsibility principle. Fields and internal methods should be private, interfaces should be used where appropriate, and abstractions should be used to reduce coupling.

Design patterns introduced in the lectures and practicals are expected to be used where they contribute to a clearer and more flexible design. In particular, the Listener pattern should be used to decouple the game logic from the user interface and networking layer, and the ModelViewController pattern should be applied to enable different user interface implementations without requiring changes to the underlying model or controller code. The overall goal is to produce software that is easy to extend and that isolates complex behaviour in well-defined modules.

Code should be written in a clean and readable style, using meaningful names and following Java conventions. You should avoid code duplication and dead code, and keep methods focused and simple. Control flow should be straightforward to follow, and responsibilities should be clearly expressed at the class and method level. Where exceptional situations arise, custom exceptions could be used to signal these in a controlled manner.

Testing Testing is an essential part of the development process and a requirement for this project. You are expected to write automated JUNIT tests that verify the correctness of your implementation. At a minimum, the core game logic must be thoroughly tested.

Unit tests should not simply call methods without checking expected outcomes. Tests must include assertions that verify correctness, that is, detect incorrect behaviour. Each individual test should focus on

one aspect of functionality to make failures easy to diagnose. In addition to targeted tests of individual rules or methods, you should include at least one test that simulates the progression of a complete game using a sequence of legal random moves. Each time this test is run, it should follow a different randomised path through the game.

The purpose of this randomised test is to ensure that gameplay remains valid regardless of how the game evolves. For example, after each move, the test could verify that it is indeed the next player's turn, or that the game state correctly reflects any tokens or pieces placed on the board, or that no illegal moves are allowed. It should also check that the game correctly identifies winning or terminal conditions when they occur and that no further moves are allowed after the game has ended. By checking these properties at each step, the test helps to detect subtle errors that may only arise during extended play or in unusual game states. Writing such a test will also help you build confidence in your implementation, as it verifies that the rules you encode are consistently enforced across a wide variety of scenarios, not just the specific cases you anticipated while writing the code.

Documentation Documentation plays an important role in conveying the structure and intent of your code. All public classes and methods in your project must be documented using Javadoc. This documentation should explain the purpose of each component, its interactions with other parts of the system, and any preconditions or postconditions associated with its behaviour.

Inline comments should be used to clarify design decisions or complex logic that is not self-explanatory from the code. Comments that merely restate what the code does are discouraged; instead, comments should focus on the reasoning behind decisions, design constraints, or non-obvious consequences.

Writing documentation during development is encouraged, as it often helps clarify the design and ensures that important considerations are captured as the software evolves. You should write Javadoc before implementing classes and methods and write comments while writing code.

Packaging for submission On the day before the functionality sign-off and the code quality and testing sign-off, you must submit your implementation as a **single ZIP file** via Canvas. The structure of this ZIP file must allow the teaching staff to build and run your programs without modification. **The project must compile without errors and include any required libraries or instructions to run the system.** Your ZIP file must contain the following items:

- `src/` All Java source files of your project, using the correct directory structure that matches your package hierarchy.
- `test/` (if applicable) The source files of your unit tests if you use a separate test sources root.
- `docs/` The exported Javadoc documentation of your project.
- `README` (in the root folder of the ZIP) This file must clearly state:
  - How to build the project and run the tests.
  - The exact classes that contain the `public static void main(String[] args)` entry point for the server, the human client, and the AI client.
  - How to start each program, including example command-line usage if applicable.
  - Any external dependencies or libraries and how they should be included.
  - See `https://www.makeareadme.com` for guidance on writing a good README.

Optional directories (include only if applicable):

- `libs/` Any non-standard libraries required to compile or run your project (as JAR files).
- `dist/` Executable JAR files of the server, client, and AI.

You may also package your project using a build tool such as Maven or Gradle. If you do so, you must include the build configuration files (`pom.xml` or `build.gradle`) in your ZIP, and the build must succeed without requiring additional configuration. All required dependencies must be declared explicitly.

5 Initial software design

The first milestone of the programming project is the initial software design. In this phase, you are expected to think through the structure of your system. While you could combine this phase with writing some code or prototype to help you think through the design, it is recommended to avoid writing too much code too soon. A clear design now will help you make informed implementation choices and avoid unnecessary

rework later. The diagrams and class list are required to make your design decisions explicit so that you can receive feedback early. **All diagrams must be created using the UTML tool.**

Your diagrams must reflect serious thought and deliberate design choices. They should show how you intend to organise your software system into components and how responsibilities are divided across classes. Your diagrams should communicate design intent at the appropriate level of abstraction: detailed enough to make your decisions clear, but not cluttered with unnecessary implementation details.

The following sections describe questions that will arise when making design decisions in your project. While all of them are relevant, you are asked to choose at least one question per diagram type and visualise your answer to it.

5.1 Class diagram

Your class diagram should focus on a specific aspect of your design and illustrate how the relevant classes interact or are structured to address a particular design decision. Use it to explore and visualise questions such as:

- How will you separate the core game logic (board, rules, game state) from networking, user interaction, and AI decision-making, so that all programs use the same game implementation through a clear interface?
- What are the main components of your server, and how will they interact to handle connections, protocol messages, queue(s), and games?
- Where do you expect to apply design patterns (e.g. MVC, Listener, Strategy), and how do they contribute to structure or extensibility?

You may create more than one class diagram if it helps visualise different aspects.

5.2 Sequence diagram

A sequence diagram should illustrate how control flows between objects or components in your system in response to a specific event. It should show the entire flow of interaction: for example, beginning with an incoming protocol message or user action and ending with the resulting messages or updates being sent. You may use your diagram to address questions such as:

- When the server receives a `QUEUE` command, which classes and methods are responsible for placing the client in a queue, matching two players, creating a new game instance, and sending `NEWGAME` messages to both players?
- How is a `MOVE` command handled? Which objects validate the move against the game logic, update the game state, and inform both players of the result (by sending `MOVE` or `GAMEOVER` messages to both players)?

Additional diagrams may be included if they help you clarify other behaviours in your system.

5.3 State machine diagram

A state machine diagram should model how one part of your system moves between well-defined states in response to events or protocol messages. Its purpose is to help you reason about the correctness of state transitions and ensure your system cannot enter invalid or undefined states. A common choice is to model the lifecycle of a client (connection) from the perspective of the server. Your diagram should address the following questions:

- Which distinct states can a client be in (e.g. connecting, handshaking, idle, queued, in game, disconnected)?
- Which events or protocol messages cause transitions between these states?
- How are unexpected events handled, such as a client disconnecting during a game or failing to complete the handshake?

Your diagram should show clear transitions from one state to another in response to explicit triggers. It is not necessary to include implementation-level detail; the goal is to ensure that your system has a consistent and well-defined lifecycle.

5.4 Activity diagram

An activity diagram should model the flow of user interaction in the human client. It is used to think through how a user moves through the application, how input is processed, and how control passes between different stages of the interface. You may use it to explore questions such as:

- When starting the client, how does the user enter the server's IP address, TCP port, and a username?
- How does the user move between different phases such as queueing and gameplay?

The diagram should make the clients control flow explicit and demonstrate that your interface will be intuitive and robust for human users. You must address how invalid inputs are handled, and how the user is guided back to a valid state.

6 Initial software design sign-off (week 8)

The purpose of the sign-off is to discuss your ideas and identify possible improvements. **The initial design must be submitted to Canvas before Wednesday 13:44.** Upload a ZIP file containing:

- A list of classes, each with a short description of its responsibility. (**required**)
- Four exported diagrams (created using UTML): at least one class diagram, one sequence diagram, one state machine diagram, and one activity diagram. (**required**)
- Any supporting drafts or notes used in developing your design.

The initial design session takes place on Wednesday of week 8. A backup session is available on Thursday only for students who attended on Wednesday with a substantive attempt, or in cases where not all groups could be accommodated on Wednesday. Completing this milestone is required in order to proceed with implementation.

7 Functionality sign-off (week 10)

The second milestone of the programming project is the functionality sign-off. The goal of this sign-off is to verify that your implementation correctly follows the communication protocol, interoperates with the reference software, and supports full gameplay with both human and AI players. This sign-off ensures that your system is functionally correct before moving on to the code quality and testing sign-off. In principle, the code quality and testing sign-off will immediately follow after the functionality sign-off.

The functionality sign-off follows a fixed procedure. You will be asked to download your submission from Canvas, to show that it compiles successfully, and to demonstrate the required functionality using the provided reference server and client. We recommend that you perform the procedure yourself ahead of time to ensure that your submission will pass. **The deadline for submitting your project is Monday 23:59.**

Sign-off procedure After downloading and compiling your submitted project, you will be asked to complete the following steps to test the functionality:

1. **Your server with the reference client**:
   You will run your server and launch the provided reference client. The reference client automatically starts multiple players, connects to your server, joins queues, and plays full games continuously until it is stopped. Your server must accept all connections, process all commands correctly according to the protocol, and remain stable while handling multiple games. Crashes, hangs, protocol violations, or incorrect game behavior results in failing the functionality sign-off. **Settings: Use at least 6 clients with a 20 ms throttle.**

2. **Reference server with your AI client**:
   You will launch the provided reference server and run your AI client against it. The AI must connect successfully, join a queue, autonomously play full games from start to finish, and correctly follow the protocol at all times. Any crash, illegal move, protocol deviation, or failure to complete a game results in failing the functionality sign-off. **Settings: Use the reference server with dummy opponent.**

3. **Reference server with your human client**:
   You will run the reference server and connect your human client. A full game will be played against the AI provided by the reference server. Your client must allow a user to join a game, list online players, make moves, see what moves are valid, and complete the game. A user should be able to play without any knowledge of the protocol or internal implementation. The client must behave correctly according to the protocol and remain stable throughout the game. If the interface is confusing, incomplete, or exposes protocol internals to the user, the functionality sign-off cannot be passed.
4. **Your server with your human client and the reference client**:
   You will run your server and launch the provided reference client as a single player. You connect your human client and after playing a few moves, you terminate the reference client. Your server must not crash, and instead, it should send a `GAMEOVER` message to the human client. Afterwards, start a new reference client as a single player and start a new game with the human client. After playing a few moves, terminate the reference server. The client must not crash, and instead, it should terminate or restart gracefully with an informative message.

Passing the functionality sign-off confirms that your system meets the core functional requirements and is ready for the code quality and testing sign-off.

8 Code quality and testing sign-off (week 10)

The code quality and testing sign-off directly follows the functionality sign-off. This sign-off verifies that both team members understand the system they have built and that core gameplay behaviour is sufficiently tested. The purpose is to evaluate whether your implementation demonstrates sound software engineering practices and whether you can explain and justify your design decisions.

Minimum requirements To be eligible for this sign-off, your project must meet the following criteria:

- The submitted project compiles and runs without errors (functionality sign-off).
- Automated tests exist for the game logic. These tests must cover normal gameplay (valid move sequences, correct turn progression) and at least some edge cases (illegal moves, win/draw conditions).
- Tests must include assertions that verify correctness. Tests that only call methods without checking expected outcomes are not sufficient.

Code deep-dive During the session, teaching staff will select parts of your code and ask each student to explain them. You are expected to understand not just the parts you personally wrote, but the entire program. Typical questions may include:

- What is the responsibility of this class, and why is it separate from other parts of the system?
- Can you explain how this method works and why it is implemented this way?
- Why did you choose this data structure (e.g., `List`, `Set`, `Map`) for managing this part of the program?
- What happens when a client joins the queue? Show me the classes involved and describe the flow.
- How is game-over detected in your code, and what happens next?
- Where is the game state stored, and how do you ensure it remains valid according to the game rules?
- Walk me through what happens when a move is received, from start to finish.
- Which test would detect a bug in this method or prevent a regression?
- Why is this method public? Could it be private or have reduced visibility?
- How do the human client and AI client reuse shared code? Which classes enable that reuse?
- What happens if an illegal move is attempted? Which part of the code handles this scenario?

Questions may range from high-level interaction between classes to beginner-level concepts such as visibility, encapsulation, or data type choices. You are not expected to memorise every line of code, but you must be able to explain and justify your choices.

Feedback and outcome The teaching staff will use the grading rubric to assess your software quality. The sign-off then concludes with feedback on your code structure, clarity, testing, and maintainability. Passing this sign-off confirms that your implementation meets the minimum standards and that you understand your own code. The feedback you receive is intended to support your growth as a computer scientist and software developer, and to help you improve in future projects.

9 Reflection sign-off (week 10)

The fourth milestone of the programming project is a short reflection session. The purpose is to reflect on what you have learned about software design, implementation, and teamwork. You will prepare a small number of slides with bullet points that will be used to guide a conversation with the teaching staff. Each student is expected to actively contribute during the session and share their own perspective.

Reflection on design Use your slides to reflect on how your design evolved from the initial plan to the final implementation. The following topics should be prepared:

- Which parts of your initial design worked as expected?
- Which parts of your initial design had to be changed, and what triggered those changes?
- Which design decisions (during the entire project) had a particularly positive or negative impact on your code quality or development process?
- If you were to start again, what one or two changes would you make to improve adaptability, testability, maintainability, or robustness? Why?
- What design alternatives did you consider but decide not to use? Why?

Reflection on process and collaboration You should also reflect on how you worked together as a team. Topics you should discuss include:

- How did you divide tasks, and ensure that both team members stayed involved in the full project?
- Which collaboration practices or tools did you use (e.g. pair programming, regular check-ins, Git branching), and how effective were they?
- What went well in your teamwork?
- What challenges did you encounter in planning or collaboration, and what would you do differently in a future project?
- What did you learn about your own working style, communication, or personal strengths?

Sign-off format The reflection sign-off is a guided conversation based on your slides. You are not expected to give a full presentation; the slides are prompts to support discussion. **Your slides must be submitted to Canvas before Tuesday 23:59.**

10 Tournament (week 10)

In Week 10, an AI tournament will be held in which the computer players developed by all teams compete against each other. Participation as a team in the tournament is **mandatory** and serves as the final sign-off of the project. A schedule will be posted before the tournament.

11 Suggestions for working effectively

Successful projects are rarely the result of last-minute effort. These practices can help you stay on track:

- **Start by reading the entire project description carefully** to understand all requirements before making design decisions.
- **Plan your collaboration using your group contract**. Agree on working hours, communication methods, and how decisions will be made.
- **Make a planning ahead of time**. Set daily goals and regularly review your progress.
- **Meet every day** to coordinate tasks, review progress, and work together. Leaving issues unresolved quickly leads to problems. It's best to be in the same room (or voice chat) as much as possible.
- **Use pair programming regularly**. It improves code quality, reduces bugs, and ensures both partners understand the full system.
- **Use Javadoc and comments as part of your thinking process**. Write them while designing and coding, not afterward. They help you clarify intent and avoid mistakes.
- **Test continuously**. Do not postpone testing until the end.

Software quality rubric

Criterion	1 (fail/poor/missing)	4 (subpar)	6 (sufficient)	8 (good)	10 (excellent)
Packaging (10%)	Fails to meet at least 2 of the criteria from **sufficient**.	Meets only 2 of the criteria from **sufficient**.	• The project builds without errors and passes all tests. • Any required libraries other than JUnit are included. • Javadoc is correctly exported and included.	As **sufficient**, and: • There is a proper README file with a description, building requirements and building instructions, testing instructions and how to run the software.	As **good**, and: • Each program (server/client/ai) is exported as an executable jar file.
Clean code (15%)	We grade this by considering six items. If an item is fulfilled, it counts as 2 points. If an item is mostly fulfilled (violated only incidentally or in a minor way), it counts as 1 point. • No violations of IntelliJ Inspections "Style problems". • Constants are used where appropriate. • No obvious duplicate or repetitive code, i.e., good code reuse. • Methods are simple and serve a clear goal. • Variables and methods have descriptive names. • No dead code (commented code or unused methods or fields).				
	4 points or fewer	5 – 6 points	7 – 8 points	9 – 10 points	11 – 12 points
Tests (25%)	One of the following applies: • Gameover conditions are not checked. • Common execution flows are not checked. • Tests are trivial (can't detect actual bugs)	There are non-trivial game logic tests that check common execution flows and gameover conditions.	• Tests cover common execution flows and edge cases of game logic, with at least 75% line coverage of methods related to moves and gameover conditions. • Tests are not trivial, i.e., can detect actual bugs.	As **sufficient**, and: • There is a test that plays a full game from beginning to end with random valid moves (each run tests a different game), testing conditions after every move. • Tests have at least 90% line coverage of game logic code.	As **good**, and: • All tests are documented with Javadoc. • Each test is simple and has a clear goal.
Encapsulation (15%)	Some fields are directly accessed from other classes.	Some fields are not private but they are not actually accessed from other classes.	All fields except constants are private.	As **sufficient**, and: • Methods are only public when intended to be used from other classes.	As **good**, and: • Other classes cannot indirectly access fields that are managed by the object, for example via getters and setters.

Continued on next page

(Continued)

Criterion	1 (fail/poor/missing)	4 (subpar)	6 (sufficient)	8 (good)	10 (excellent)
Structure (20%)	Fails to meet any of the criteria from **sufficient**.	Meets only 1 of the criteria from **sufficient**.	• There is a clear separation of concerns between most classes. That is, a few classes may have no clear purpose or multiple responsibilities. • There are multiple clearly defined components, reflected in the package structure. The structure is reasonable.	As **sufficient**, and: • Each class exhibits high cohesion, meaning it has a singular, well-defined responsibility, and related functionalities are logically grouped together within the class. • The network protocol handling can be easily replaced without affecting other code.	As **good**, and: • Interfaces are applied where appropriate to ensure low coupling between components. • There is clear and correct application of at least two design patterns (Listener, Strategy, Factory, Builder, Chain of Responsibility, MVC) using interfaces.
Javadoc (15%)	Fails to meet any criteria from **sufficient**.	Meets only one criteria from **sufficient**.	• Most classes and interfaces have reasonable class-level Javadoc. • Most methods of the game logic have reasonable Javadoc.	• All classes and interfaces have reasonable class-level Javadoc. • Most methods have reasonable Javadoc.	There is reasonable Javadoc on all classes, on all methods, and on all packages.
	Reasonable Javadoc is: a concise and complete description for other programmers what a class is for or what a method does (in relation to the parameters), and possible return values/exceptions. Parameters, return values and exceptions should also be documented.				