|  |
| --- |
|  |

**University of Twente**

HEX PROJECT 2023

**REPORT**

**Alexandru Verhovetchi s2958716**

|  |
| --- |
|  |

1. **Introduction**

In the vast realm of board games, Hex stands out due to its strategic depth, elegant design, and rich history. Originating in the 1940s, Hex is played on a hexagonal grid, typically of size 11x11 or 13x13, where two players compete to form an unbroken chain of their own pieces connecting their respective sides.

Despite its seemingly simple rules, Hex unfolds into a complex and often profound game of strategy and tactics. The goal of this project was to delve into the mechanics, mathematics, and computer strategies of the Hex game.

This report provides a comprehensive overview of our exploration, encompassing game design, algorithmic strategies for gameplay, user interface development, and performance evaluations.

By journeying through each of these facets, we aim to not only deepen our appreciation for Hex but also to foster its continued growth and enthusiasm among both new and seasoned players.

Whether you are a board game enthusiast, a researcher, or simply curious, we invite you to traverse this journey with us as we dissect, re-imagine, and celebrate the wonders of Hex.

**2. Table of Contents:**

[1. Introduction 1](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.c5utj7y11oft)

[2. Table of Contents 2](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.wm14ck5djw18)

[3. Stakeholders 3](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.wtm8u5qn6y2g)

[4. Functional Requirements 4](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.tz35l8afv581)

[5. Class Diagrams 5](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.lf5lo8ds50yh)

[6. Class and Responsibilities List 10](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.i5uwg31h2xwn)

[7. Testing Plan 13](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=kix.85g908ik2dlr)

[8. Testing Strategy 15](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.vk645r3p3j5l)

[9. Test Report 18](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.bgv6h89mr797)

[10. Metrics and Conventions 24](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.lxim1uepuy25)

[11. Reflection on Initial Design 26](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.z5dewyf78sa2)

[12. Reflection on Final Design 26](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.k9x4qujgapoq)

[13. Concurrency Mechanism 27](https://docs.google.com/document/d/1FXr3GlBVDKxBIgNXcTBf1e-t2nOs039ENopLOsAkzGA/edit" \l "bookmark=id.yrybdm9aw0l2)

**3. Stakeholders:**

Project Team: This group, comprising the student undertaking the project, is entrusted with the game's creation. Their role involves adhering to the project directives and upholding coding norms and best practices.

End Users: These are the primary beneficiaries of the project, the ones who'll be immersing themselves in the game. Their interaction predominantly lies with the game's functionalities.

Tournament Organizers: As the individuals overseeing the tournament setup and execution at the module's conclusion, these organizers have a vested interest in the project. Their focus will be on assessing the AI's operational efficiency and the integrated features.

Fellow Students: These are other learners in the same course, each crafting their unique client-server games. They are essential stakeholders since the game in development should be equipped to interface and vie with their respective game systems.

IT Department or Server Management: The server custodians, entrusted with managing the game-hosting servers, have a stake in the project. This is because the developed game must align with the infrastructure in place and fulfill any stipulated prerequisites.

**4. Functional requirements:**

1. Game mechanics: The game should be able to support standard Hex rules:

*- Hexagonal grid of size 9x9.*

*- Two players, traditionally colored red and blue.*

*- Player 1's goal is to connect the top and bottom sides, while Player 2's goal is to connect the left and right sides.*

*- Form a path of your pieces connecting the opposing sides of the board marked by your colors before your opponent does.*

*- Starting: Player 1 (usually red) begins by placing a piece anywhere on the board.*

*- Turns: Players alternate turns, placing one piece on any unoccupied space.*

*- Swap Rule: After Player 1's first move, Player 2 may either place a piece or change the color of Player 1's first piece to their own color.*

*- Immutability: Once placed, pieces cannot be moved or removed.*

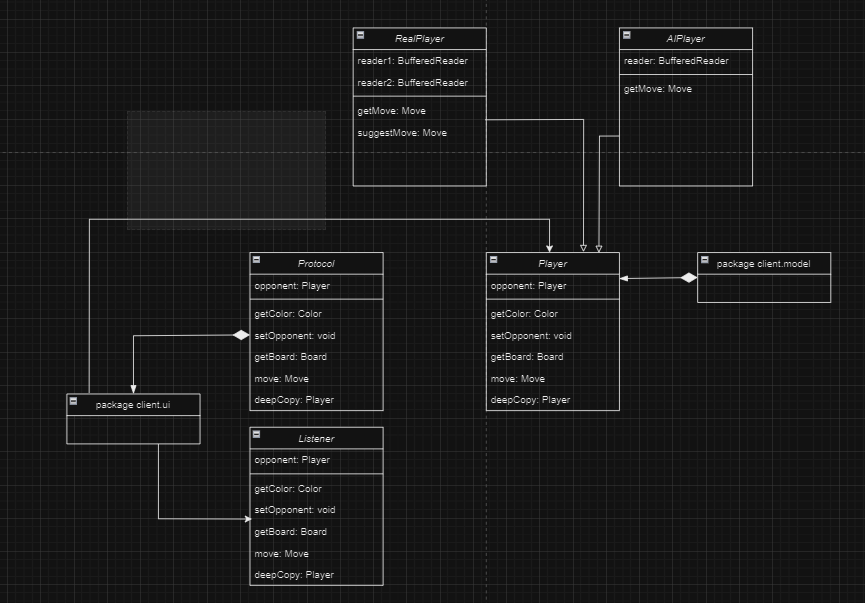
*- The game ends when one player connects their two sides.*

*- Draws are impossible: the board will eventually fill, and it is impossible for both players to complete their paths simultaneously.*

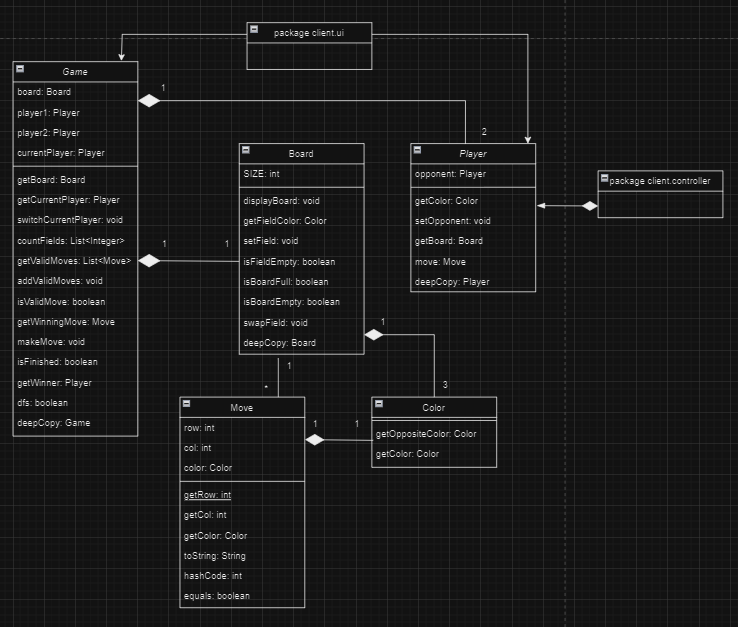
*- If the board is filled and one player has not connected their path, the other player must have done so.*

1. When the server is started, it will ask the user to input a port number where it will accept connections. 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 valid move as a hint via TUI.
4. The client can play a full game automatically as the AI without intervention by the user.
5. 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.
6. All communication outside of playing a game, in particular the handshake and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.
7. Whenever a client loses connection to a server, the client should gracefully terminate.
8. Whenever a client disconnects during a game, the server should inform the other client(s) and end the game, allowing the other player to start a new game.

**5. Class diagrams:**



Package client.controller



Package client.model

Client part class diagram description:

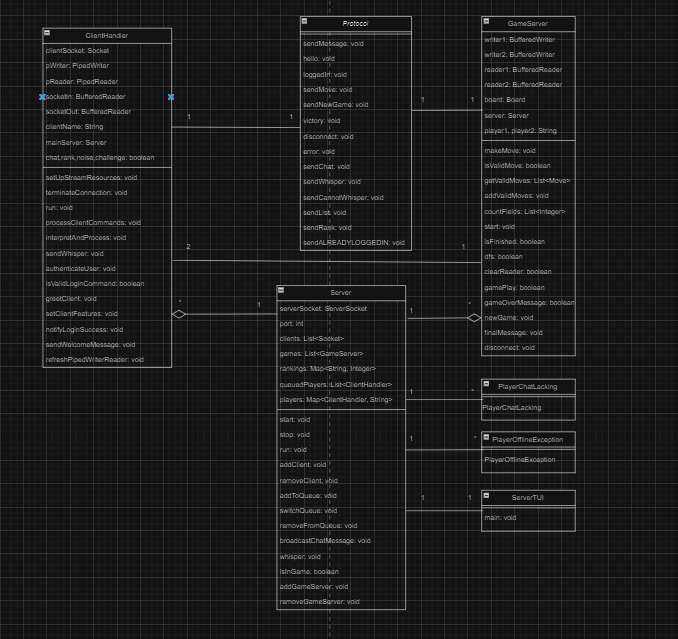
The primary class in this system is the TUI, which offers users the option to select the server and type of the player that will play. It also enables users to issue commands to the server.

Within the game package, there are classes that depict the game elements: the Board class symbolizes the game's play area and its current status; the Color class is for individual colors set by players during gameplay; and the Move class encapsulates the player's action and the consequent board's status.

Moving on to multiplayer mode, the class diagram comprises a Listener class that's responsible for catching server commands and player classes designated for multiplayer participants. The purpose of the Listener class is to monitor server instructions, whereas the player classes denote each participant in the multiplayer setting.

To ensure seamless communication between the Listener and the multiplayer players, a piped reader is in place. This ensures that information is promptly exchanged between the server and players, making the multiplayer gameplay interactive and updated in real-time.

Summarizing, the class diagram efficiently outlines the different classes and how they interrelate in the text-based game setup. This clarity in roles and interactions of each class streamlines the game development process.

Package server

Server part class diagram description:

The server-side of the design revolves around the ServerTUI class, which serves as a TUI to initiate a new Server, specifying the port number to be used. This class alerts whenever a client connects or disconnects from the server.

Central to the system is the Server class, representing the crux of the server functionalities. It manages all client interactions, transmits necessary messages to them, and oversees the server's queue list to decide the commencement of a game round.

Next, we have the ClientHandler class, designed to receive client commands and provide a fitting response, channeling these commands to the actual game. It shares a close relationship with the Server class as it bridges the client's requests with the server's responses, ensuring each query is processed accurately.

The game's essence lies within the GameServer class, which shoulders all the gaming logic. This is the reason the ClientHandler class maintains an association with the game logic - to accurately execute client's requests. Notably, there's an aggregation relationship between the Game class and the Server class. In this bond, the GameServer is reliant on the server for its functioning. The server lays down the essential infrastructure and resources, like networking and data handling, upon which the game server thrives.

In conclusion, the Server class is integrated with custom exceptions, aiding in handling specific errors or misinterpretations in various situations. Also, both the ClientHandler and GameServer are connected to the Protocol class, ensuring messages are structured and sent in accordance with the guidelines laid down in this class.

**6.** Class and Responsibilities List:

Client Controller package:

**AI** *(AI player that uses an random algorithm to determine the best move)*

**RealPlayer** *(Human player that reads the moves from the console)*

**Listener** *(Listens to the server and writes the messages to the piped writer if it is related to the game else it will print)*

**NetworkPlayer** *(Player that reads its move from the server -> Listener)*

**Protocol** (*Sends the messages to the server according to protocol)*

Client Model package:

**AbstractPlayer** *(Abstract Player that implements Player interface)*

**Board** *(Creates Hex board of the game)*

**Game** *(The Game class this is class has all the logic for the game)*

**Color***(Contains all colors that can be used in the game)*

**Move** *(Represents functionality of a move for the board)*

**Player** *(Interface that represents a player in the game)*

Client View package:

**ClientTUI** *(The main class of the client application that is responsible for starting games and for connecting to the server/handling commands from the user)*

Sounds package:

**Sound** *(Class that handles the sounds of the game)*

Server Controller package:

**ClientHandler** *(Listens for commands from the client and sends back the appropriate response or forwards the command to the game)*

**Protocol** *(Sends the messages to the writer according to protocol)*

**Server** *(The Server class this is the server class that handles all the clients’ connections)*

Server Model package:

**GameServer** *(The Game class this is the game class that handles all the game logic)*

Server View package:

**ServerTUI** *(The Main class this is the main class that starts the server)*

Exceptions:

**PlayerChatLacking** *(The PlayerChatLacking class is the exception that is thrown when a user does not have chat extension)*

**PlayerOfflineException** *(The PlayerOfflineException class is the exception that is thrown when a user is offline)*

Explanation of packaging structure:

The system is bifurcated into two primary packages: the server and the client. The server package is concerned with back-end operations such as processing requests, managing and updating player rankings, and dispatching responses. On the other hand, the client package emphasizes the front-end, focusing on data visualization, capturing user interactions, and forwarding requests to the server.

Both these primary packages are further segmented in line with the Model-View-Controller (MVC) design paradigm. Within this framework:

- The model package houses classes central to game logic, taking on the duty of managing and altering game data.

- The view package is vested with the task of presenting a user interface, ensuring that players have a visual representation of game data and progress.

- The controller package acts as the orchestral, mediating user interactions and bridging the gap between the model and view, ensuring seamless coordination.

Adopting this MVC-based architecture brings clarity and structure to the system. Each package gets a distinct responsibility, which in turn simplifies code maintenance and modifications. Furthermore, this modular approach boosts flexibility. For instance, changes in the view can be executed without altering the underlying game logic (model), and vice versa. Additionally, any enhancements or additions to features can be localized to the specific segment, minimizing disruptions to other parts of the system. This structure not only streamlines development but also aids in isolating issues, thereby making debugging and testing more efficient.

**7.** Testing Plan:

1. Testing Timeline:

**Unit Tests:**

**Duration**: 1 week

**Objective**: Examine individual elements of the game - methods, classes, game mechanics, server operations, and AI processes.

**Goal**: Validate each unit’s performance and ensure its functionality aligns with design specifications.

**Component Integration Tests:**

**Duration**: 1 week

**Objective**: Analyze interactions between game's diverse components, such as client-server communication.

**Goal**: Validate integrated component functionality and address potential issues.

Holistic System Tests:

**Duration**: 2 weeks

**Objective**: Evaluate the game's overall behavior and functionality as an integrated entity.

**Goal**: Confirm the game performs optimally when all elements converge.

**User Acceptance Tests:**

**Duration**: 1 week

**Objective**: Gauge game's usability, accessibility, and user experience with actual players.

**Goal**: Determine if the game aligns with user requirements and is primed for deployment.

1. **Roles & Responsibilities:**

**Alexandru**:

**Unit Testing (client-side):** Focus on client aspects like game mechanics, UI, and AI. Draft and implement test cases.

**Unit Testing (server-side):** Direct focus on server operations, especially communication with the database. Create and run test scenarios.

**Integration Testing:** Work on evaluating the interaction between client-server and client-AI.

**Acceptance Testing:** Facilitate user testing by sourcing participants, overseeing test sessions, and interpreting results.

**System Testing:** Concentrate on the comprehensive evaluation of the game, assessing behavior and performance metrics.

1. Expected Outputs from Tests:

Post each testing stage, testers are anticipated to produce:

**Test Results Snapshot:**

A concise report noting test case outcomes, execution dates, tester details, and relevant comments.

**Performance Metrics Overview:**

A detailed account of test results, defect count, and test coverage extent.

**User Acceptance Summary:**

Feedback from users focusing on game usability, accessibility, and user experience insights.

**System Testing Synopsis:**

Chronicles of actions taken during system testing, outcomes, and status updates.

**Comprehensive Test Analysis:**

Our team is committed to presenting a conclusive test analysis encapsulating all test results, data metrics, and executed activities. This comprehensive document will shed light on individual test nuances, particularly user acceptance facets, and offer a deep dive into the software's overall quality. This documentation is vital to ensuring continuous improvement and readiness for the next development phases.

**8.** Testing Strategy:

**Introduction**:

Our systematic testing approach serves as a road-map detailing how our software will undergo evaluation. This strategy illustrates the distinct testing phases, required resources, testing schedules, and managerial aspects of our testing procedure. It aligns with the project's overarching objectives and priorities.

**Scope & Boundaries:**

The testing scope for our system, encompassing server and AI components, encompasses the server's operations, game dynamics, AI operations, performance metrics, error resolution, digital security, and network interactions. By extensively evaluating these sectors, we aim to ensure that the Hex game operates seamlessly and satisfies its users. Our testing regimen includes broad system and unit tests, simulating potential real-world scenarios. This covers identifying potential vulnerabilities and evaluating the system's resilience and performance during high server traffic. However, it's worth noting that complete protection remains elusive, as predicting every malicious user activity or simulating vast user numbers remains a challenge.

**Test Phases:**

1. Unit Testing: These tests zero in on individual game components like game mechanics, server operations, or AI processes. They allow for efficient error detection and rectification at a granular level. Essentially, these tests ensure comprehensive system coverage.

2. Integration Testing: With an emphasis on inter-component interactions, these tests examine communication pathways within the game, e.g., client-server or AI-game state integrations. This ensures cohesive game performance, error identification, and the accurate portrayal of game states. A high code coverage metric for this phase ensures smooth component interactions.

3. System Testing: These tests appraise the game’s end-to-end functionality. Assessments include game component loading, ensuring legal player/AI moves, and the game's capacity to host multiple players and games. These results gauge the game's current efficiency, stability, and areas needing enhancement.

4. Acceptance Testing: Centered on user experience, these tests evaluate system usability and accessibility. Performed by actual users or stakeholders, this phase aims to ensure that the software aligns with user requirements, thus preparing it for a real-world environment. Feedback, especially regarding user-friendly features, will guide necessary system adjustments.

**Testing Compatibility:**

System and automated unit/integration tests collaborate as follows:

System tests aim to assess the entire system, encompassing all its elements, their interactions, and interconnected dependencies. For our system, such tests emphasized replicating multiplayer game settings, involving several clients, a central server, and even applying pressure tests on the server.

Conversely, automated unit and integration tests target the verification of discrete components and how they interface with one another. Specifically for our system, integration tests mimicked AI vs. AI gameplay, while unit tests validated the fundamental integrity of game operations.

Explanation on the least-tested components of our system:

Automated testing has its limitations, particularly where user input is indispensable. Nonetheless, it remains a crucial facet of our Hex game testing. Automated methodologies permit rapid component and interaction checks, streamlining bug detection and rectification. Yet, some areas, like console commands or move entries requiring user input, lean more towards manual testing. Although not ideal, hands-on gameplay sessions substantiate the operability of these components. We're confident in their efficacy, bolstered by multiple gameplay sessions that flagged no issues. Similarly, while game board console display wasn't subjected to automated tests – given its visual validation necessity – manual checks vouched for its accurate and satisfactory performance.

**9.** Test Report:

**1. Unit testing:**

The main goal is testing individual functions and components of the game, such as specific methods or classes that describe the game logic, server functionality and AI functioning. In order to ensure that each function of our project is functioning correctly, we provided the following test classes:

**TestClass AbstractPlayerTest** - tests class AbstractPlayer and verifies the correctness of its methods. Therefore, there are some examples of the tests that were provided to check that: testGetColor(), testSetGetOpponent() that test the correct setting of an opponent to a player and mark assignment.

**TestClass AITest** - completely tests all methods of the AI class, verifying correct functionality based on expected results. Therefore, there are some examples of tests that were provided to ensure correct implementation of the AI logic : testGetColor(), testSetGetOpponent(), testMove() that test AI’s algorithm and logic correctness.

**TestClass BoardTest** - tests correct initialization of the board as well as evaluating the board’s completion or other different scenarios. Therefore, there are some examples of tests that were provided to ensure correct implementation of the Board object: testSetFieldColor(), testIsBoardFull() that test the Board’s modification, checking its fields etc.

**TestClass ClientHandlerTest** - tests correct functionality of the current ClientHandler class that is responsible for handling commands from the client and forwarding them to the actual gameplay. Therefore, the following test was provided to ensure correct command handling of the server: testClientHandler() that verifies simulating different types of inputs from a client.

**TestClass GameServerTest** - tests game functionality, game end detection, validation of the board. Therefore, there are some examples of tests that were provided to ensure correct game logic functionality: testIsGameOver(), testGetWinner(), testGetValidMoves() that test overall game rules and correct validation of moves during different scenarios.

**TestClass GameTest** - has the same scope as TestClass GameServerTest but is destined for Game Class from the client package. Therefore, there are some examples of tests that were provided to ensure correct game logic functionality: testIsFinishedr(), testGetWinner(), testGetValidMoves() that test overall game rules and correct validation of moves during different scenarios.

**TestClass HumanPlayerTest** -tests the creation of a human player as well as the usage of the methods that it acquires. Therefore, the following test was provided to ensure correct functionality: testMove() which tests the correct performance of the getMove() method.

**TestClass ListenerTest** - tests correct functionality of the Listener class responsible for listening to the server and providing messages to the player. Therefore, the following test was provided to test the overall functionality of the class: a testListener() that checks if different messages are read correctly and then sent to the game.

**TestClass ColorTest**- tests correct functionality of the methods related to the Color class. The following tests were provided to ensure correct functionality: testGetOppositeColor() and testGetColor() which test if correct colors are returned in different scenarios.

**TestClass MoveTest** - tests correct functionality of the methods related to the Move class. Therefore, there are some examples of the tests that were provided to ensure correctness of the Move class: testHashCode() that checks if a correct index is created.

**TestClass NetworkPlayerTest** - tests creation of an NetworkPlayer object from the NetworkPlayer class that is responsible for reading moves from the server to the Listener class. The following tests were provided to ensure correct functionality of the class: testSetGetOpponent(), testGetColor() that test the correct setting of an opponent and color assignment.

**TestClass ClientTUI**- tests correct functionality of the Client class. Therefore, the following test was provided to ensure correct implementation: testClient() that tests overall Client class performance.

**TestClass ProtocolTest** - tests that verifies if the actual results correspond to the expected outputs and if they are sent correctly to the server. Therefore, there are some examples of tests that were provided to ensure correctness of Protocol class: testHello(), testSendUsername(), that test if sending username and a hello message is correct and can be read from the server.

**TestClass ServerProtocolTest** - tests the server protocol by verifying if the actual results correspond to the expected output. Therefore, there are some examples of tests that were provided to ensure correctness of the ServerProtocol class: helloTest(), sendMoveTest() that test if certain messages are sent correctly.

Explanation of how unit tests cover each other’s gaps:

Each test class focuses on testing a specific class or component in the system, and its methods verify the correctness of the class's behavior in different scenarios. These tests cover each other’s gaps and ensure that the system works as expected in all diverse situations.

For example, the TestClass AbstractPlayerTest verifies the methods of the AbstractPlayer class, but it does not cover the interaction between AbstractPlayer and other classes in the system. This gap is covered by other test classes, such as TestClass AITest, which tests the AI class, which extends AbstractPlayer, and verifies its interaction with other components. Moreover, TestClass ClientHandlerTest verifies the functionality of the ClientHandler class, but it does not test the interaction between ClientHandler and the game server. This gap is covered by TestClass GameServerTest and TestClass GameTest, which test the game server and the game logic, respectively.

Consequently, each unit test possesses a unique and subtle scope in testing of the game functionality, but all together they complete each other to ensure overall correctness of the system.

**2. Integration testing:**

The primary objective of this phase is to validate the synergy between various components of the system. Using JUnit 5, we simulated the gameplay to ensure seamless integration between these components. The key test classes associated with integration testing include:

**TestClass GameServerTest** featuring the test:

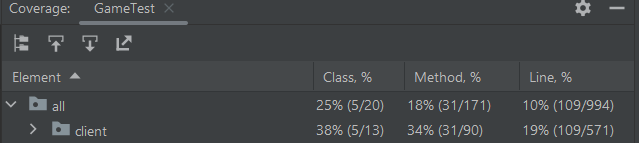
testRandomGame() - This examines the all-encompassing functionality during a random game simulation. Though primarily designed for the GameServer class, it invokes methods from auxiliary classes. This ensures that both game mechanics and functionalities mesh well.

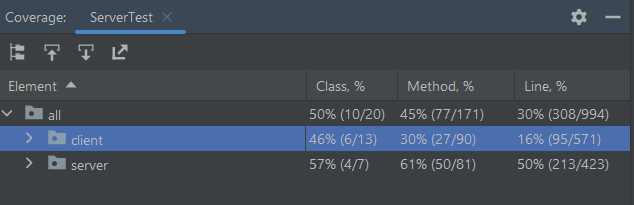
**TestClass GameTest** offers the test:

testGetWinner() - It assesses the full-scale functionality throughout a spontaneous game simulation. Aligned with the Game class, this test integrates and employs methods analogous to the previously mentioned test.

**TestClass ServerTes**t presents the test:

testPlayGame() - This confirms the server's capacity to support two clients executing multiple games without glitches. It accomplishes this by connecting to the server, initiating hello and login commands, subsequently running 100 games, and verifying the conclusion of each game by discerning a victor.





**10.** Metrics&Conventions:

During the system testing phase, our primary objective was to produce a robust system built upon quality code. To realize this objective, we pursued several key milestones:

1. **Code Coverage**: We aimed for around 90% coverage during testing to ensure most parts of the code were executed and verified.

2. **Structural Integrity**: We emphasized the importance of components having minimal dependencies (low coupling) while ensuring that each component was focused and efficient in its function (high cohesion).

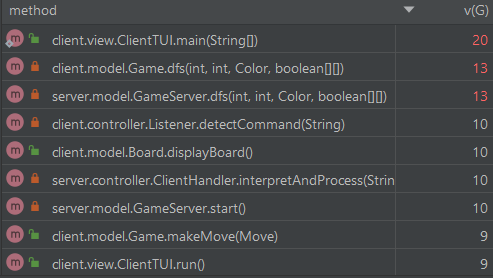
3. **Defect Density**: We aimed to reduce defects to near zero for every thousand lines of code, showcasing code quality.

4. **Test Success**: Our ambition was for 95% of test cases to pass, reflecting a system that operates as designed.

5. **User Acceptance**: We aspired for an 80% approval rating from users, indicating that our system met user requirements and expectations.

6. **Performance Benchmarks**: Key indicators such as AI move-response time and server responsiveness were set to specific targets, with the AI aimed at making moves within a 10-second window.

7. **Documentation Quality**: We prioritized comprehensive documentation, striving for every piece of our code to be accompanied by Java-doc and JML for clarity and future modifications.



As a result, the project contains only few methods that have a v(G), known as level of complexity, that demonstrates a high value. A higher v(G) indicates a piece of code that may be harder to maintain, test, and understand due to its branching. While some complexity is inevitable given the diverse functions the project must perform, our consistent adherence to best practices has ensured that such complexities remain exceptions rather than norms.

This proactive approach not only promotes code readability and maintainability but also reduces potential risks associated with bugs that often hide in more convoluted code structures. When a method does exhibit a high v(G), it has been subjected to rigorous testing and peer review to guarantee its robustness and reliability.

Moreover, to aid in future updates or possible refactoring, methods with elevated complexity have been thoroughly documented. This ensures that any developer, whether they were part of the initial team or new to the project, can understand the rationale and logic behind the method's structure.

In summary, while our project does have areas of higher complexity, these are approached with caution, thorough testing, and clear documentation, underlining our commitment to quality and long-term sustainability.

During the developmental phase, adhering to established conventions is pivotal in ensuring high-quality code and a user-friendly system. The core conventions we are committed to upholding throughout the project's development include:

1. **Code Conventions:** Emphasizing clarity, we'll employ intuitive naming conventions for variables, methods, and classes. Comments will be concise and relevant, ensuring a uniform coding style across the board. Also, we'll avoid overly long methods or classes to maintain readability and simplicity.

2. **Directory Structure:** A well-organized directory and package structure will be maintained. This will facilitate easier navigation, making it straightforward for developers to understand and locate different components of the system.

3. **Version Control:** Leveraging platforms like GitLab, we'll employ version control mechanisms. This ensures synchronized work among developers, tracks changes efficiently, and fosters effective communication throughout the development cycle.

4. **Access Modifiers:** Proper use of access modifiers will be crucial. We'll use 'private' or 'protected' for methods not meant for external use, preserving the system's integrity. Overuse of 'public' methods will be avoided both for minimizing potential errors and bolstering security.

Incorporating these conventions ensures a consistent development approach, fosters collaboration among developers, and augments the overall quality and security of the system. It's essential to regularly review and update these practices in line with evolving best practices and project requirements.

**11.** Reflection on initial design:

The foundational design of the Hex TUI game, inclusive of its server and client components, was largely indicative of the concluding system, albeit with a few requisite modifications to its design blueprint.

On the client's end, several classes were introduced to incorporate more features, a development we could somewhat foresee based on our past familiarity with the game dynamics. Even with these adjustments, the primary client design remained largely consistent and aptly represented the game's overall mechanisms.

Conversely, the server's design witnessed a more significant overhaul. Certain anticipated classes became redundant and were eschewed in favor of novel classes that better suited the evolved design. Challenges in preempting the exact server functionalities led to a departure from our preliminary server design. Classes such as Board, Player, HexThread were deemed superfluous. Instead, the introduction of classes like ClientHandler, GameServer, and ServerTUI enriched the system's design, ensuring it was more in line with the actual server operations.

Yet, in spite of these design shifts, our team managed to stay on course with the testing timeline, proficiently executing all the planned testing phases, spanning unit to acceptance testing. Furthermore, the implementation of the MVC design pattern fortified the system's adaptability, simplified component testing, and delineated class roles, imparting more clarity and structure to the project. Such refinements underscore the eventual design's efficacy and our team's nimbleness in implementing vital changes for a triumphant project realization.

**12.** Reflection on final design:

Our team is committed to delivering a seamless and efficient system that provides a smooth user experience, devoid of glitches or unforeseen exceptions. However, recognizing the perpetual nature of advancement, we understand that there's always scope for refinement and augmentation in any system.

Looking ahead, one of our primary ambitions is to enhance our testing framework. By broadening our test coverage and delving into edge cases, we aim to bolster the system's reliability and ensure its performance remains impeccable.

Furthermore, we aspire to streamline our code-base. Refactoring specific methods and classes will not only enhance code clarity but will also pave the way for easier modifications down the line. Introducing bespoke exceptions is another avenue we're exploring. Custom exceptions can enhance system resilience by alerting users in a more intuitive manner when issues arise.

Lastly, the user interface is always an evolving facet of any system. With the aim of elevating the user experience, we're considering the integration of animations, enhanced graphics, and other visual elements. Such additions will not only beautify the game but also make it more engaging, drawing in both seasoned players and novices alike.

**13.** Concurrency Mechanism:

**Shared objects:**

Connected users

Player ranks

Queue players

Games played

**Threads:**

Server

Client handler

Game thread

Sound thread

Listener

TUI thread

In essence, the server-side application architecture is built on multiple threads catering to a variety of tasks. These encompass ClientHandler threads for each active client, a distinct thread overseeing server management, and numerous threads to manage individual game sessions. Each game thread is crucial, taking charge of game logic and, crucially, updating player ranks. A potential snag here arises when scores are added to ranks. Due to the concurrent nature of these threads, simultaneous access to modify ranks can trigger race conditions.

This concurrency concern extends to other aspects too. For instance, the game list — a comprehensive list of ongoing games — is shared among all threads. Whether it's ClientHandlers verifying the existence of a game for a user, a game concluding and removing itself post the final communication, or the server enlisting a game when there's an adequate player count queued up, race conditions are a potential threat. The list of connected users is another shared resource that’s prone to concurrency issues, especially during simultaneous connect and disconnect events or when many users express interest in joining the queue.

Our remedy for this intricate situation is the strategic implementation of synchronization within the server thread. This synchronization zeroes in on the specific object in the cross-hairs, ensuring serialized access, thereby eliminating the risks of concurrent modifications.

The client facet of the application has its own set of threads — the game listener and the TUI — which are interconnected via a piped reader. Ingeniously designed to avoid simultaneous data access, this model inherently prevents race conditions. But it isn't foolproof; there were scenarios where rapid game conclusions led to IOException errors from the piped reader. Our solution was a judiciously placed try-catch block to manage these exceptions, ensuring uninterrupted TUI operations.

Upon initiating a connection to a server, a game listener springs into action. Simultaneously, the TUI, activated right from startup, takes charge of visual representation and user input processing. When a user kickstarts a game via the TUI, a dedicated game thread takes the baton, orchestrating game logistics and syncing with server updates.

Lastly, for an immersive experience, there's a temporary thread dedicated to audio cues. Spawned momentarily, its sole purpose is audio playback, keeping it isolated from shared data and, by extension, race conditions.