FINAL PROGRAMMING PROJECT REPORT

By Mira Gozbenko and Danil Badarev, s3187152 and s3210928

Group: Normal-26

Explanation Realized Program Design

A list of classes, each accompanied by a description summarizing its main responsibilities

Game model related classes:

- Game interface – outlines the main methods that should be implemented within a game, making it reusable for multiple games with different rules and mechanics.
- GameDnB – implements the logic of the Dots and Boxes game specifically, contains the board and keeps track of players
- BoardDnB – represents the board within a Dots and Boxes game, implements the functions needed for game rules of the game, in fact, implements the main part of functional methods
- Components – represents the parts of which the board consists (vertical line, horizontal line, dot) and the colors of the players
- Move interface – represents the move in a random game, for reusability purpose
- MoveDnB – implements the move in a Dots and Boxes move specifically, contains the colour and location of the move
- NotValidMove exception – exception to handle the execution if invalid move is being done
- Player interface-the interface representing the player in a random game (reusable)
- AbstractPlayer abstract class – implements Player, needed for representing a player in a game
- HumanPlayer-class that implements the methods needed for a person to play the game

Game UI related classes:

- DnBTUI-executable, responsible for playing a game, either with a player or an AI

Game AI related classes:

- StrategyLVLEasy-represents the tactics for the AI player, the move is being done randomly if assigned to ComputerPlayer
- StrategyLVLNormal- represents the tactics for the AI player, the move is done such that if there is a box AI can close-it will
- StrategyLVLHard- represents the tactics for the AI player, the move is done such that another player can't finish the box in the next move if possible if given to ComuterPlayer class.

Related to server:

- Protocol – protocol with final variables, dictates the protocol used to communicate between the server and the client
- Server – the server with executable main method, responsible for running the server, accepting the incoming connections. Handels the messages from the client according to the protocol, exept LOGIN
- ClientHandler-class, representing the client on the server. Responsible for containing the user's information and parsing the commands between the Server and the ServerConnection classes. Keeps the track of the game instance and the handshake start status.
- ServerConnection-accepts and sends the messages, parses them to the Server class. LOGIN is being processed in ServerConnection and sent from here.
- SocketServer – abstract class for implementing the server logic

Related to client:

- ClientConnection-communicates with the server, parses the messages in the protocol format over the network,
- ClientTUI-executable, creates the instance of client TUI, shows the game the user is playng, parses the protocol messages to the server depending on client input
- Client-represents the client on the client side, parses the messages between the clientTUI and the ClientConnection, holds the game instance on the client side
- ClientListener interface-used to create a ClientTUI
- SocketConnection- abstract class for implementing the communication over the network

Related to testing:

- GameDnBTest- extensivelly tests the game logic of the program, you can find the existing tests either in the program or in the overall testing section where the pictures of the coverage metrics results are provided
- BoardDnBTest-tests the whole board class with 100% coverage metrics

An overview of the organization of the code into classes and packages, with a design rationale that explains the structure's logic, specifically addressing separation of concerns and the breakdown into simpler components

We have the following packages, each of them is responsible for its part of the game:

The main one, which contains the rest of the packages

The packages inside it:

1. **server** package – separates the server part of the application
2. **client** package – separates the client part of the application
3. **test** package – separates the tests the tests from the rest of the application
4. **game** – responsible for the game logic, contains 3 packages:
    a. **model** – package with the game logic
    b. **ui** – package with the user interface
    c. **ai** – package with the ai player and its difficulties/strategies

This structure, which separates four different parts of the software and divides the game into three main branches, ensures the separation of concerns on a package level.

On the class level, the classes in the game package usually serve one simple purpose and the breakdown into simpler components is done properly: this is visible in the previous section, where each class in the game package has its specific purpose. Class Contents is responsible for containing the parts of the board while DnBGame implements the rules, DnBBoard implements the rules, DnBMove implements move, and overall, the names of the classes are very straightforward are represent their responsibilities.

However, the separation of concerns within the packages "client" and "server" can be improved: in server and client part of the application the classes serve several purposes, which is visible in the previous section.

For example, ClientHandler parses the messages and keeps track of the game instance on the server side and Client class represents the client on the opposite side, parses the messages between the clientTUI and the ClientConnection and holds the game instance on the client side, which is not the instance of elegant separation of concerns and can be improved.

Concurrency Mechanism

List of the most important threads and their purpose

We didn't create customized threads, however, the server implements the following ones:

Thread **main** in the Server class, responsible for starting the server and accepting the incoming connections

Threads that are being invoked in Serever class by calling **serverConnection.start()**, which are responsible for handling incoming messages, processing them and sending outcoming messages back to the users.

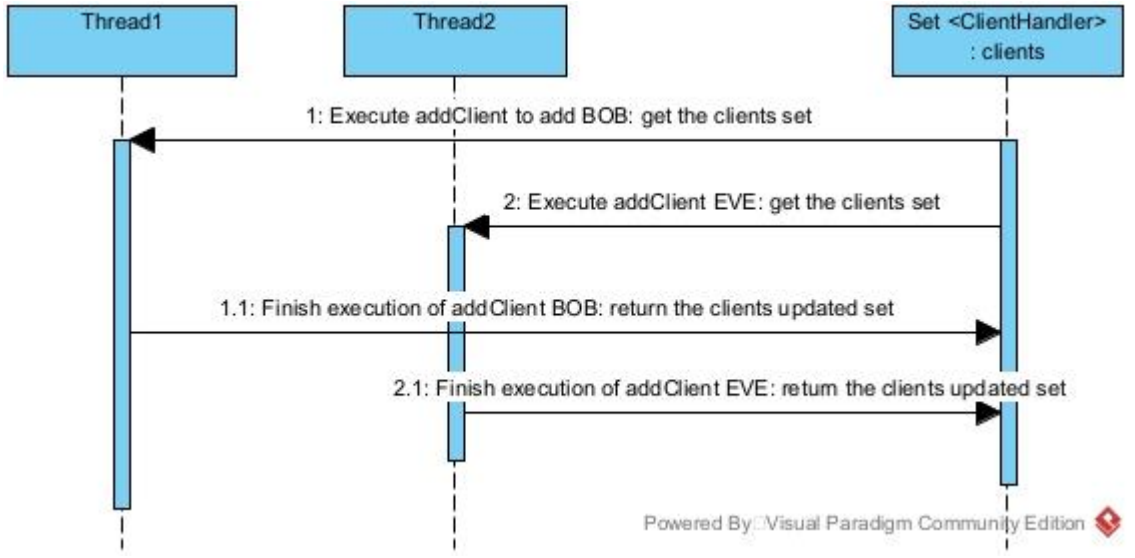
The most significant shared objects, detailing why each object is shared and by which threads

There are several objects shared by threads: set of client handles clients and list of client handlers queue in the Server class, as multiple **serverConnection.start()** threads should be able to access them when they want to remove client, add client, or add him to the queue or delete from there.

Identification of at least one potential race condition, explanation of the problem and where it might occur, along with an example of a possible consequence if the race condition is not handled.

The race condition can occur in the Server class when the threads try to call addClient(Client handler) to add him to the clients set. They both take the fields clients initial state, one of them puts it back with a new client first, but the second one overwrites it only with it's one. That's how one of the clients might be not added to the set.

The solution for this problem is making addClient(Client handler) method in the server connection synchronized. This will create a lock on the method when it is called by one thread and another one won't be able to access it.



In the example above, BOB won't be added to the list due to the race condition.

Reflection on design

Overview of the initial design

(The initial design document will be provided below the overview for the grading convenience.)

Strengths of the initial design: The class diagrams of the project components provide the base for building the game and the server/client side. Moreover, the plan for connecting the game to the server side is present with initial plan on doing the AI with different levels of difficulty. The plan for creating threads is

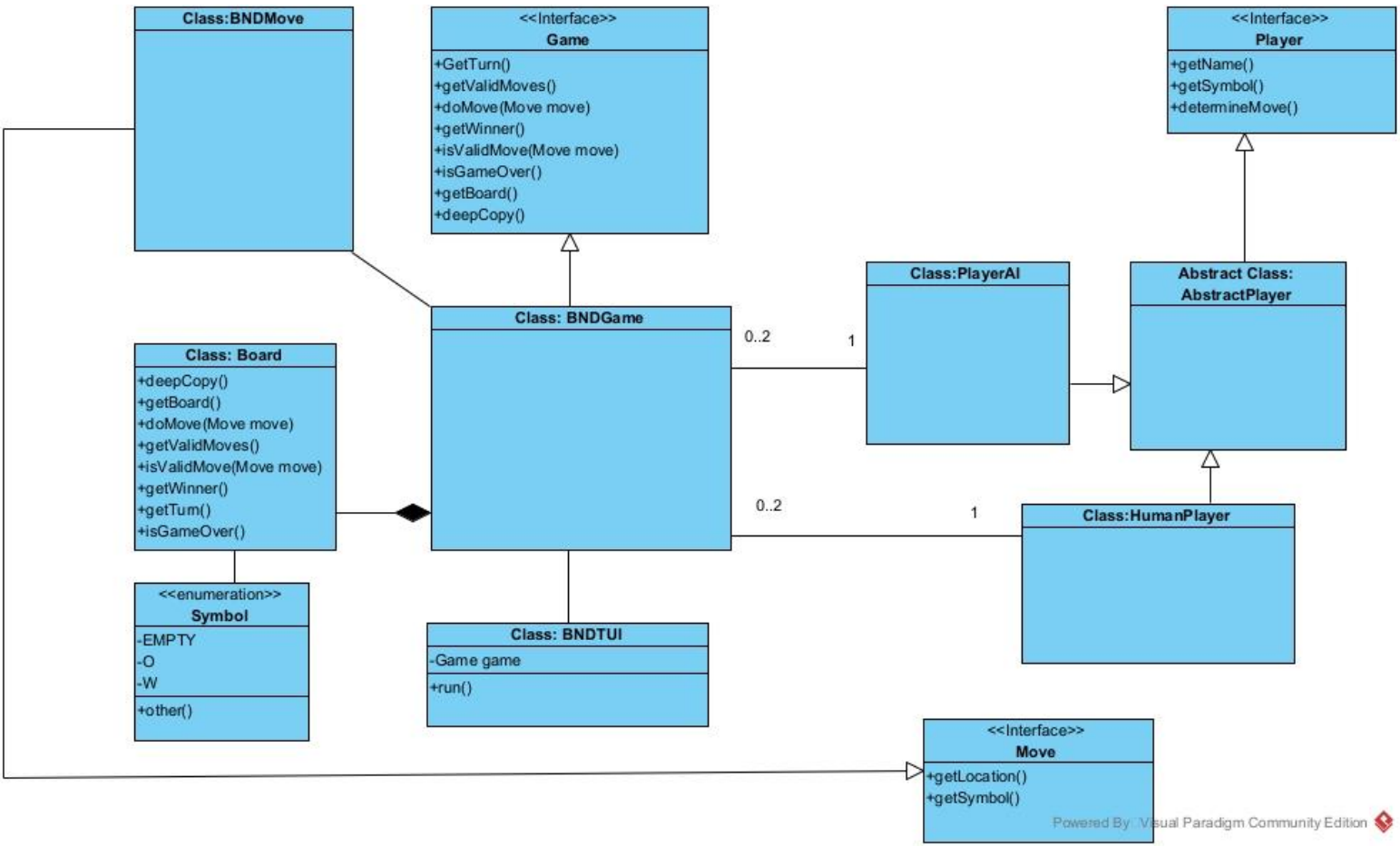
reasonable and logical, which should have helped in implementation of the multiple connections on the server.

Weaknesses: The sequence diagram doesn't indicate the interaction between the server and the clients if one of them disconnects. The game class diagram poorly incorporates the way to indicate the player's move: placing different letters to indicate different players on the board, which is not user friendly and poorly visually distinguishable. The class diagram doesn't indicate how should the game be connected to the client side of the application.

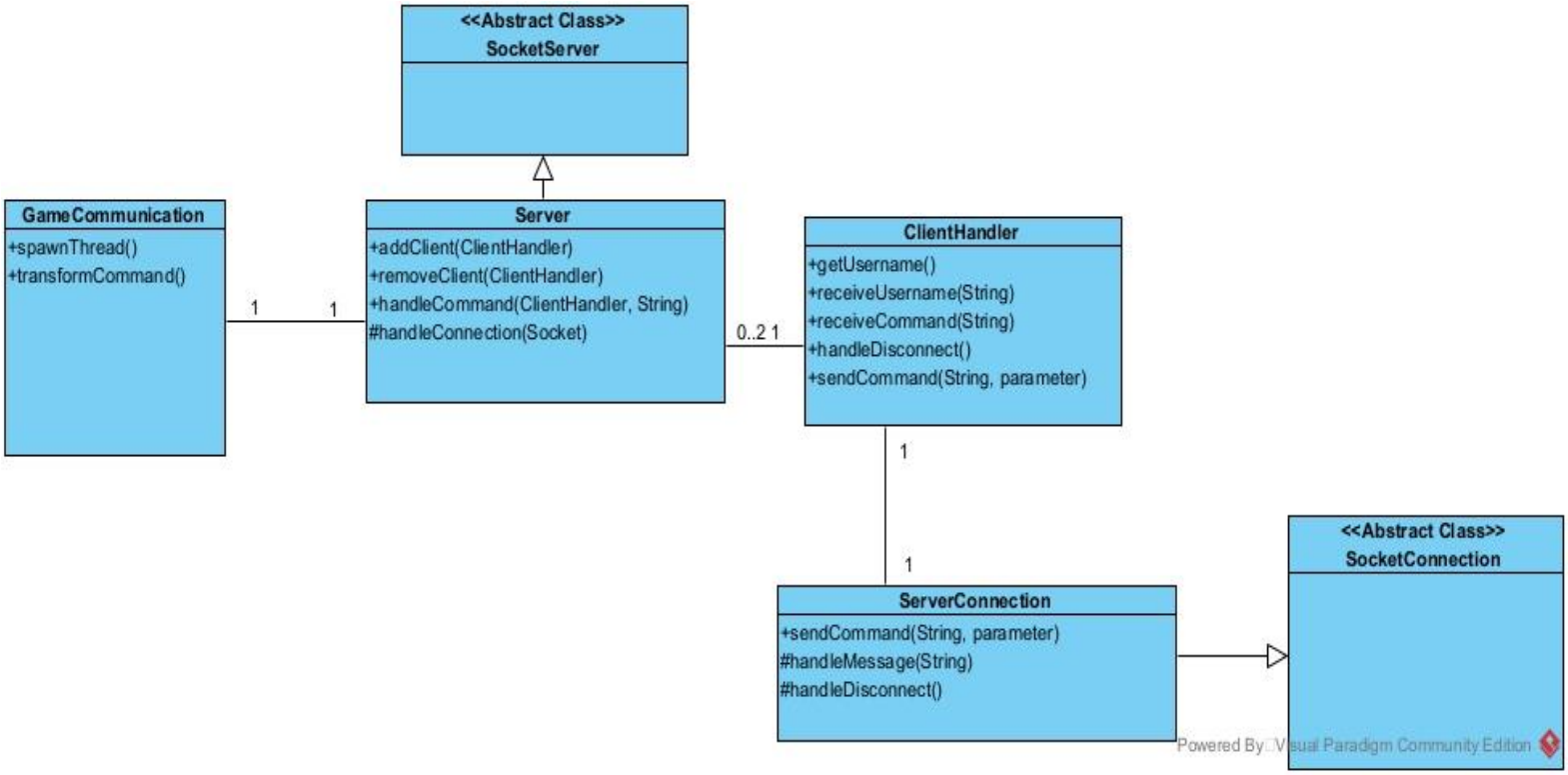
Overall, the initial design provides developers with the idea how the server-client interaction will be performed, with starting plan on building several levels of AI. It also gives the direction to the implementation of the game connection to the server and gives an overview on the plan where to run threads and what is their purpose, but has its own minuses, as gaps in the implementation strategy and affects user-friendliness.

Initial design of the programming project (for the overview of the initial design)

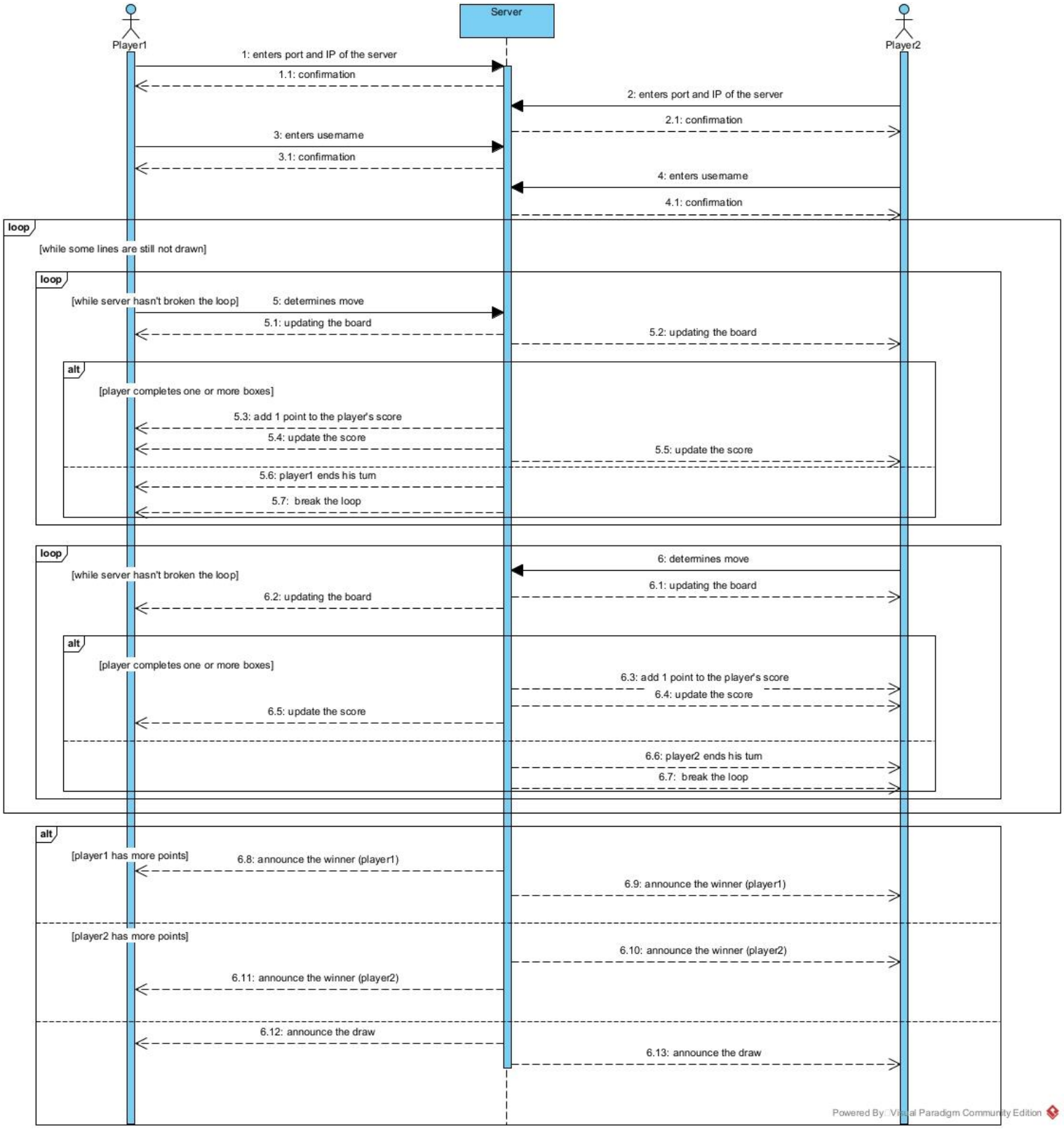
Partial initial design class diagram for the game part of the application:


Class:BNDMove
<<Interface>>
Game
+GetTurn()
+getValidMoves()
+doMove(Move move)
+getWinner()
+isValidMove(Move move)
+isGameOver()
+getBoard()
+deepCopy()
<<Interface>>
Player
+getName()
+getSymbol()
+determineMove()
Class:PlayerAI
Abstract Class:
AbstractPlayer
Class: BNDGame
0..2
1
Class: Board
+deepCopy()
+getBoard()
+doMove(Move move)
+getValidMoves()
+isValidMove(Move move)
+getWinner()
+getTurn()
+isGameOver()
0..2
1
Class:HumanPlayer
<<enumeration>>
Symbol
-EMPTY
-O
-W
+other()
Class: BNDTUI
-Game game
+run()
<<Interface>>
Move
+getLocation()
+getSymbol()
Powered By: Visual Paradigm Community Edition

Initial design for the server part of the application:



Initial design sequence diagram for the players-server interaction:



Multithreading usage plan:

We will spawn threads in the DNBGame class responsible for the execution of the game logic, threads will execute the progression of the game in parallel, including player moves and updates to the game state. Moreover, the threads will be used in GameCommunication class (which connects the server with the game part of the program) to handle player actions simultaneously if they are trying to execute the move/another type of input at the same time. The threads will be despawned when players disconnect or the game session is finished/server is shut down.

AI implementation plan:

We are *planning to develop an AI system that offers three distinct levels of difficulty for an enhanced gaming experience. This AI client program is designed to connect to a dotsandboxes game through user-provided parameters, including port, IP address, username, and desired difficulty level. At the easy difficulty level, the AI player will employ a naive strategy, making random moves without a predefined plan. Moving up to the medium difficulty level, the AI player gains an understanding of the game's winning conditions. If there are any moves on the board that score points, the AI player will make this move; otherwise, it will function as an easy mod AI. The hardest difficulty allows AI player to determine if his move might grant the opponent an opportunity to score points. The AI won't play any of these moves unless it has no choice. In all other aspects, the AI at the hardest difficulty level behaves similarly to its medium-level alternative. This diverse approach ensures an engaging and adaptable gaming experience for users.*

Explanation of the main or most important changes made from the initial design to the final design

The first big decision was to create the class with all the board parts instead of the Symbol class. This implementation allowed us to change the parts of the board later during the implementation process. Moreover, it improved user-friendliness, as we changed the characters (letters) we intended to use from the start to actual colours. This insured the

Second decision was to remove the GameCommunication class because of the rush before the project submission, it's main responsibilities were transferred to the BnDTUI, Server and client. Now, the instance of the DnBGame is running on the serever and both clients after receiving the NEWGAME protocol message from the server.

The third change was not implementing our own Throwable thread objects, again, because of the lack of time.

The rest of the program seems to be the same as the initial design, except the small changes in naming the classes.

Discussion of key decisions and justifications for them during the project's evolution

The first key decision, before the submission of the initial plan, was to take the initial implementation of the Tic-Toc game, ChatServer and the ChatClient we made during the practicals, as the code recycle is a good practice, which helps in implementation, as it is easier to recycle the code you wrote and already understand than write it from scratch.

The second key decision was to change the design of the connecting the game to the server and the client. Initially, the idea was to connect game only to the server, so that clients don't even need to have the code of the game to paly it. However, as soon as the TA told us about another way to do it. We could create the instances of the game on both two clients and the server. We changed it, as it was actually easier to implement and we quickly grasped the general idea, unlike with our initial plan.

The third key decision was to refuse from the idea of doing the AI of the highest difficulty recursive. We started implementing it, however, we understood it was not rational to implement it with our logic. It should have done recursive game scan and do the most beneficial move. Unfortunately, it would take too much time for it to calculate all the possible game instances, so we returned to the more feasible idea of it identifying the move that would not let the opponent to do the move that would close the box, until it is possible.

Overall testing strategy

Explanation of the test plan of the game logic:

Objective:

The goal of the test plan is to verify that the game logic works correctly in order to ensure a smooth gaming experience.

Scope:

The test plan covers the testing of the game logic, focusing on core functionalities related to player interactions, scoring, box completion, turn handling, and any other critical gameplay-related aspects.

Testing approach:

- Unit testing to ensure the isolated verification of the small program parts and functions. (Validation of the player move, box completion logic, decision whose turn is it, etc.)
- Integration testing to ensure that parts of the program work correctly together (Example: the whole game instance test)

Chosen coverage metrics: Line coverage

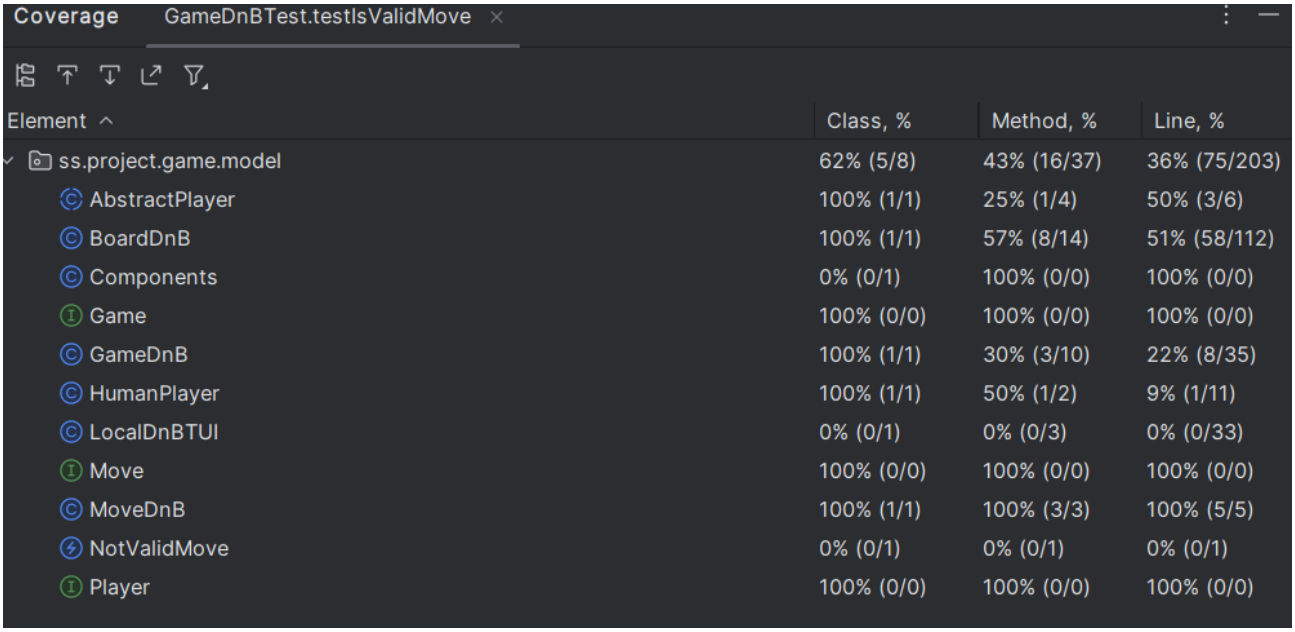
Incorporation of coverage metrics:

Coverage metrics used to determine the reliability of the game in our implementation are Line Coverage: which percentage of lines is being covered during the testing, which indicates what percentage of lines of code is reliable in at least one execution. In the case of a part of the code with higher cyclomatic complexity, more unit tests will be provided to ensure the reliable functionality.

Chosen complexity metrics: Cyclomatic complexity

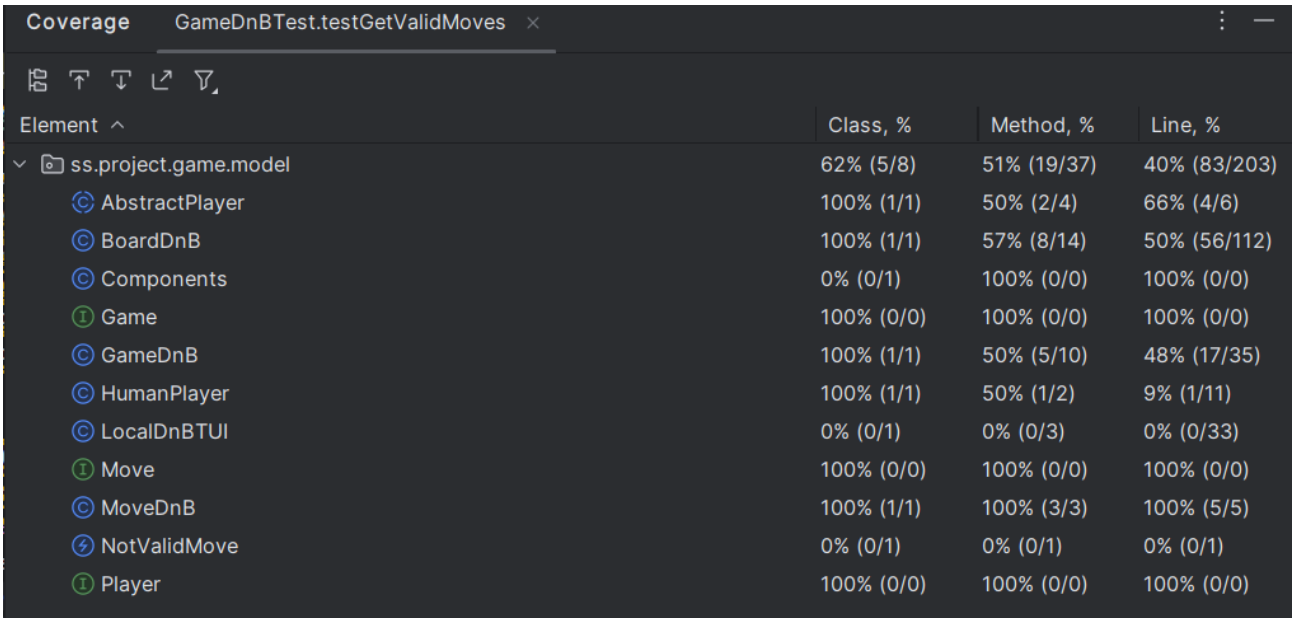
Tested code parts, with visual evidence of code coverage metrics:

Testing isValidMove() function



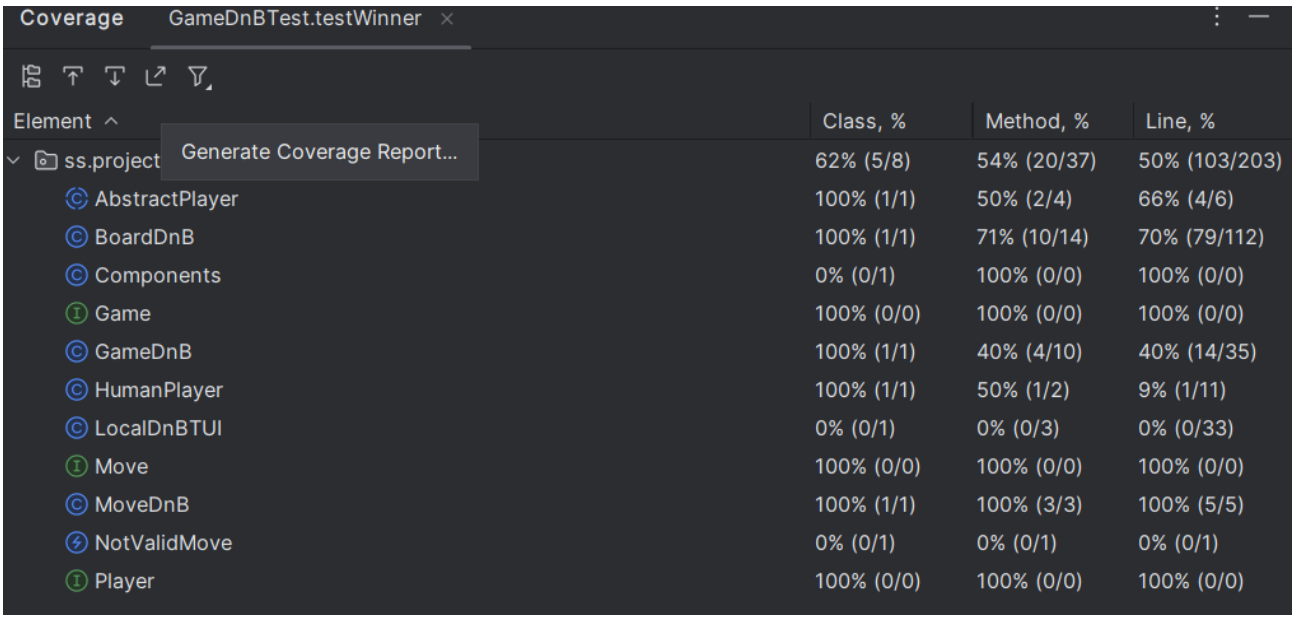
Element	Class, %	Method, %	Line, %
ss.project.game.model	62% (5/8)	43% (16/37)	36% (75/203)
AbstractPlayer	100% (1/1)	25% (1/4)	50% (3/6)
BoardDnB	100% (1/1)	57% (8/14)	51% (58/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	30% (3/10)	22% (8/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Testing getValidMoves()



Element	Class, %	Method, %	Line, %
ss.project.game.model	62% (5/8)	51% (19/37)	40% (83/203)
AbstractPlayer	100% (1/1)	50% (2/4)	66% (4/6)
BoardDnB	100% (1/1)	57% (8/14)	50% (56/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	50% (5/10)	48% (17/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Testing getWinner() function



Element	Class, %	Method, %	Line, %
ss.project	62% (5/8)	54% (20/37)	50% (103/203)
AbstractPlayer	100% (1/1)	50% (2/4)	66% (4/6)
BoardDnB	100% (1/1)	71% (10/14)	70% (79/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	40% (4/10)	40% (14/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Testing getTurn

Coverage GameDnBTest.testGetTurn

Element	Class, %	Method, %	Line, %
ss.project.game.model	62% (5/8)	43% (16/37)	40% (83/203)
AbstractPlayer	100% (1/1)	50% (2/4)	66% (4/6)
BoardDnB	100% (1/1)	57% (8/14)	58% (65/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	20% (2/10)	22% (8/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Testing of the deep copy of the game

Coverage GameDnBTest.testDeepCopy

Element	Class, %	Method, %	Line, %
ss.project.game.model	62% (5/8)	51% (19/37)	44% (91/203)
AbstractPlayer	100% (1/1)	50% (2/4)	66% (4/6)
BoardDnB	100% (1/1)	64% (9/14)	59% (67/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	40% (4/10)	40% (14/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Testing of the gameOver() method

Testing of the one game from its start to its end (playGameDim2)

Overall game testing:

Coverage GameDnBTest

Element	Class, %	Method, %	Line, %
ss.project.game.model	75% (6/8)	86% (32/37)	77% (158/203)
AbstractPlayer	100% (1/1)	75% (3/4)	83% (5/6)
BoardDnB	100% (1/1)	100% (14/14)	99% (111/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	100% (10/10)	100% (35/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	100% (3/3)	100% (5/5)
NotValidMove	100% (1/1)	100% (1/1)	100% (1/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Overall board testing:

Coverage BoardDnBTest

Element	Class, %	Method, %	Line, %
ss.project.game.model	12% (1/8)	37% (14/37)	52% (106/...
AbstractPlayer	0% (0/1)	0% (0/4)	0% (0/6)
BoardDnB	100% (1/1)	100% (14/1...	94% (106/1...
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	0% (0/1)	0% (0/10)	0% (0/33)
HumanPlayer	0% (0/1)	0% (0/2)	0% (0/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	0% (0/1)	0% (0/3)	0% (0/5)
NotValidMove	0% (0/1)	0% (0/1)	0% (0/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Test whether the identifying wrong move works correctly:

Element	Class, %	Method, %	Line, %
ss.project.game.model	75% (6/8)	40% (15/37)	31% (63/203)
AbstractPlayer	100% (1/1)	50% (2/4)	66% (4/6)
BoardDnB	100% (1/1)	35% (5/14)	39% (44/112)
Components	0% (0/1)	100% (0/0)	100% (0/0)
Game	100% (0/0)	100% (0/0)	100% (0/0)
GameDnB	100% (1/1)	40% (4/10)	25% (9/35)
HumanPlayer	100% (1/1)	50% (1/2)	9% (1/11)
LocalDnBTUI	0% (0/1)	0% (0/3)	0% (0/33)
Move	100% (0/0)	100% (0/0)	100% (0/0)
MoveDnB	100% (1/1)	66% (2/3)	80% (4/5)
NotValidMove	100% (1/1)	100% (1/1)	100% (1/1)
Player	100% (0/0)	100% (0/0)	100% (0/0)

Approach to automated and manual testing discussion:

Initially, to ensure the valid functionality of the program, the manual testing is used. As testing might get complicated with the increasing functionality, the automated testing will become more important. Manual testing can't be properly replaced while extensively testing the security and user-friendliness of the game, but can help in testing the most common security issues and parts which should be tested with a big coverage area.

Two key test case examples and description of how and what potential bugs are detected when changes to the software are made:

1.Test case: Closing the box

Description: Let player finish a box, and verify that the game reacts to it and gives the same player the turn.

Potential bugs if the changes are made: Program does not recognize the completed box or allows another player to continue.

2.Test case: Placing a line

Description: Do a move and verify that the line was correctly placed on a board with the color of the current player and the other line colors were not corrupted.

Potential bugs if the changes are made: Line is not placed, line has the wrong color/is being placed in a wrong place (or both).

Identification and discussion of edge cases handled by the tests, and identification of the potential unhandled edge cases:

Edge cases:

Completion of the box:

Edge Case: Create a scenario where a player can complete a box by doing a move to check if the game correctly recognizes and handles a box completion.

Draw Scenario:

Edge Case: Create a situation where no more boxes can be created, and both players have an equal number of completed boxes to verify that the game correctly recognizes a draw.

Edge of the Grid Moves:

Edge Case: Make moves at the edges and corners of the grid to ensure that the game correctly handles moves at the grid's boundaries without errors.

Potential unhandled edge cases:

Playing the game with different board sizes

Edge case: Play the game with different parameters of the grid in order to ensure there are no unexpected errors

Completion of two boxes at the same time:

Edge Case: Create a scenario where a player can complete two boxes by doing a move to check if the game correctly recognizes and handles a box completion.

Identification of concrete challenges faced in testing, limitations in the current testing strategy, and evaluates the effectiveness of the testing strategy in detecting errors, such as unexpected results, bugs, and crashes.

Challenges and limitations:

The challenge of the current strategy using manual testing is that there is extensive amount of the code that has to be tested. Writing unit tests for each single method might take a lot of time and effort. Moreover, due to the complexity of the program, it is challenging to test every single possible branch, another words, player or AI behavior can result in unpredictable outcomes, which is challenging to cover in unit and integration tests. Limitations may arise when it comes to trying to find visual inappropriateness of the system or to simulate the player behavior, as it is unpredictable, which leads to limit of how we can handle edge cases.

Evaluation of effectiveness:

The unit testing is efficient while trying to detect errors, while integration testing is crucial for catching unexpected results due to the integration of the components. Similarly, the unit testing can be used for catching in-function bugs and errors and integration testing can be used to verify whether the using units together causes bugs and errors, even though the testing of the edge cases depends on the amount and extensiveness of the tested parts of the code.

Discussion of confidence in least-tested components.

In our instance the least tested component is getValidMoves() function which is not being tested yet. It is needed for the AI and will be tested on upcoming stages of implementation. The rest of the program is either being tested in unit tests or integration tests.

Proposals and explanations of improvements of the testing strategy, reflecting on aspects like adaptability, efficiency, or robustness/security, and explains how these improvements would positively affect the testing process

Implement User Acceptance Testing to involve users in testing, allowing for the discovery of unexpected errors, which will make a testing process more efficient.

Expand the amount of test cases that will help to identify more errors/bugs, which will extend the testing process' impact on program reliability.

Extend the testing with a simplified UI/UX check. This will lead to possible identification of glitches, which can't be covered by unit or integration tests, which will make the testing more extensive.

Do an adaptability testing by running the program on different OS (Linux), which will directly increase adaptability. This will reveal potential errors that might come up during the attempt to run the program on different operating system.

Do an automated regression testing: it will detect if the fixes of the code lead to the performance drop. This will allow to control the program's performance during the testing.

Integrate the security scanning tools. Those will automatically identify security vulnerabilities in the game code during the testing.

Reflection on process

Individual reflections from all group members on initial planning and collaboration

Individual reflections are based on the document we sent for the 3rd carrier skills assignment - planning, which included division of the project into work packages, estimation of the time frames and division of the work within the team. The 3rd carrier skills assignment document is provided below the reflections for the verification and grading convenience.

Danil's reflection on initial planning and collaboration:

The initial planning provided me with foundation for our project, helping me to understand our tasks and responsibilities. The division of work into Work Packages (WPs) and tasks gave me an overview of the working plan, helping me to see the project's trajectory. The mitigation strategies for potential risks, such as requirements ambiguity and unexpected technical problems gave me the confidence that the team is ready to address difficulties, though the problems with creating the networking part still arose. Nevertheless, I was responsible for tasks related to design patterns, class diagrams, and sequence diagrams, and detailed timeline and milestones provided me with a structured framework. The collaboration plan, which included regular team meetings and open discussions, ensured a communicative and collaborative work environment, which helped us to write the functional server and client even with a time shortage. Before, while we were writing the game, the incorporation of pair programming during the development phase showed its effectiveness in complex problem solving. Therefore, the initial planning made a path for the project and contributed to its final submission.

Mira's reflection on initial planning and collaboration:

The initial planning document provided us with an approach to teamwork during all project execution phases. The emphasis on pair programming during the development phase played an essential role in problem-solving and knowledge sharing, moreover, the use of online and in-person meetings, along with tools like Discord, provided us with a flexible and accessible communication platform. The distribution of tasks based on individual strengths ensured that both my partner and I contribute to the project effectively while keeping track of the time left for the particular part to develop. We have to admit that we had problems with initial planning, as we underestimated the importance of the time for writing the server and client. Even though, the decision to work on weekends only when necessary was a good boarder, which allowed us to take into consideration our health and personal needs. Even while responsibilities were divided, the collaboration plan encouraged open communication and regular updates, which constantly contributed to smooth workflow, even in times like the server or client building. Categorizing the tasks into groups (necessity, quality, deception, and waste of time) helped in prioritizing activities based on their impact on project success. Overall, the collaboration plan committed to effective teamwork and communication, creating an environment to productive collaboration, while the initial planning outlined the individual tasks and necessary timeframes, even though several of them weren't estimated properly.

The 3rd carrier skills assignment – Planning (needed for ''initial planning and collaboration'' and "general task division" parts context)

Agreed desired grade – 8.7
Agreed on no extra points/bonus parts.
WP1 Title: ***Design of the project system***
Timeline: Start Date: 17 Jan 2024- End Date: 20 Jan 2024
Description: The Design work package outlines the fundamental elements of the project, focusing on defining the program's overall structure and implementation plan. This involves selecting an appropriate design pattern, discussing the software requirements, and creating diagrams to guide the development process.
Tasks:
- Task 1: Discuss and pick a reliable design pattern and discuss the software requirements

- Task 2: Create a class diagram of the upcoming project
  - Task 2.1: Create a game class diagram
  - Task 2.2: Create a server-client related class diagram
- Task 3: Build a sequence diagram of the user connection to the server

Dependencies: T3 depends on T2.2
Resources:
Human Resources:
Danil Badarev: task 1, task2, task 3
Mira Gozbenko: task 1, task 2, task 3
Material Resources: 2 PCs, Visual paradigm, Discord
Risks:
Requirements ambiguity: requirements ambiguity could result in additional work needed for construction, which can affect the schedule/timelines.
Resource constraints: unforeseen issues with the availability or functionality of the material resources (PCs, software) may influence progress
Communication breakdown: miscommunication between team members during the design pattern selection and diagram creation could result in the need for rework, impacting the timeline.
Unexpected technical problems: Complexities in the game design or server-client interaction may arise, which will require additional time to design.

Mitigation strategy:
Requirements ambiguity: Schedule additional sessions for requirement clarification, in our case – on Sunday
Resource Constraints: Maintain regular checks on the condition and functionality of the PCs and software. Have a plan such as an access to backup equipment.
Communication breakdown: Implement regular team meetings and status updates to ensure everyone is on the same page.
Unexpected technical problems: Set aside additional time with no plans on it to settle unforeseen technical problems, in our case, this can be done on Sunday.
Collaboration: Open discussion during the meeting in person and via the Discord video call, if necessary. 2 hours will be spent on the task 1 on Jan 17, 10 hours (5 on 2.1 + 5 on 2.2) on task 2 on Jan 18 and 3 hours on task 3 on Jan 19. The work will be done during the practicals and, if necessary, additional meeting will be scheduled. By default, the Sunday is a day of no working on the project. The work after 20.00 should be done only in case of unexpected circumstances. No work will be done during the lunch break.
Progress Tracking:
- Milestone 1: Completion of the task 1
- Milestone 2: Completion of the task 2.1
- Milestone 3: Completion of the task 2.2
- Milestone 4: Completion of the task 3

Notes: The tasks should be performed together to avoid miscommunication and improve awareness of the overall task and the system design.
Recognizing Achievements: Every milestone will be kept in mind and support a positive mindset of the developers.

WP2 Title: ***Development part***

Timeline: - Start Date: 22 Jan 2024- End Date: 28 Jan 2024

Description: The cycle of program development is related to implementation of the system with respect to the agreed design. Development part includes building a game, creating a server system and connecting them, it should be possible for multiple users to play online. Additionally, it includes building an AI player. Developing part WP is being performed simultaneously with the Testing WP after the Jan 24th.

Tasks:

- Task 1: Build the game logic
- Task 2: Implement the network communication
  - o Subtask 2.1: Implement the protocol
  - o Subtask 2.2: Implement the server side
  - o Subtask 2.3: Implement the client side
- Task 3: Implement a user interface
- Task 4: Build an AI player

Dependencies of tasks of WP2: tasks 3 and 4 can be started only after the task WP2-1, T1 depends on WP1-T2.1, T2 depends on WP1-T2.2

Resources:

Human Resources:

Danil Badarev - Task 2, 4

Mira Gozbenko – Tasks 1, 3, 4

Material Resources: 2 PC's, \\:https\gitlab , IntelliJ IDEA 2023.2.5, Discord

Risks:

AI building time consumption: building an AI player might consume more time than indicated, as this task is the most complicated.

Requirements ambiguity: requirements ambiguity could result in additional work needed for construction, which can affect the schedule/timelines.

Resource constraints: unforeseen issues with the availability or functionality of the material resources (PCs, software) may influence progress

Communication breakdown: miscommunication between team members during the design pattern selection and diagram creation could result in the need for rework, affecting the timeline.

Unexpected technical problems: Complexities in the game design or server-client interaction may arise, which will require additional time to implement after changing the design.

Mitigation strategies:

AI building time consumption: In case of not being able to finish on time, additional hours will be spent on Sunday Jan 28th.

Requirements ambiguity: Schedule additional sessions for requirement clarification

Resource Constraints: Maintain regular checks on the condition and functionality of the PC's and software. Have a plan such as an access to backup equipment.

Communication breakdown: Implement regular team meetings and status updates to ensure everyone is on the same page.

Unexpected technical probems: Put aside time with no plans to solve unforeseen technical problems, in our case, this can be done on Sunday.

Collaboration: strong pair programming as well as individual work, version control usage (gitlab), every day - verbal reports of the work done. 3 days – parallel work on networking and the game itself individually, 3 days starting from Jan 26 – implementing an AI together + start working on tests. The online communication in case of necessity will be performed via Discord.

Progress Tracking:

Milestones: 1 – finish task 1, 2 – finish task 2, 3 – finish task 3, 4 – finish task 4

Recognizing Achievements: Every milestone will be kept in mind and support a positive mindset of the developers.

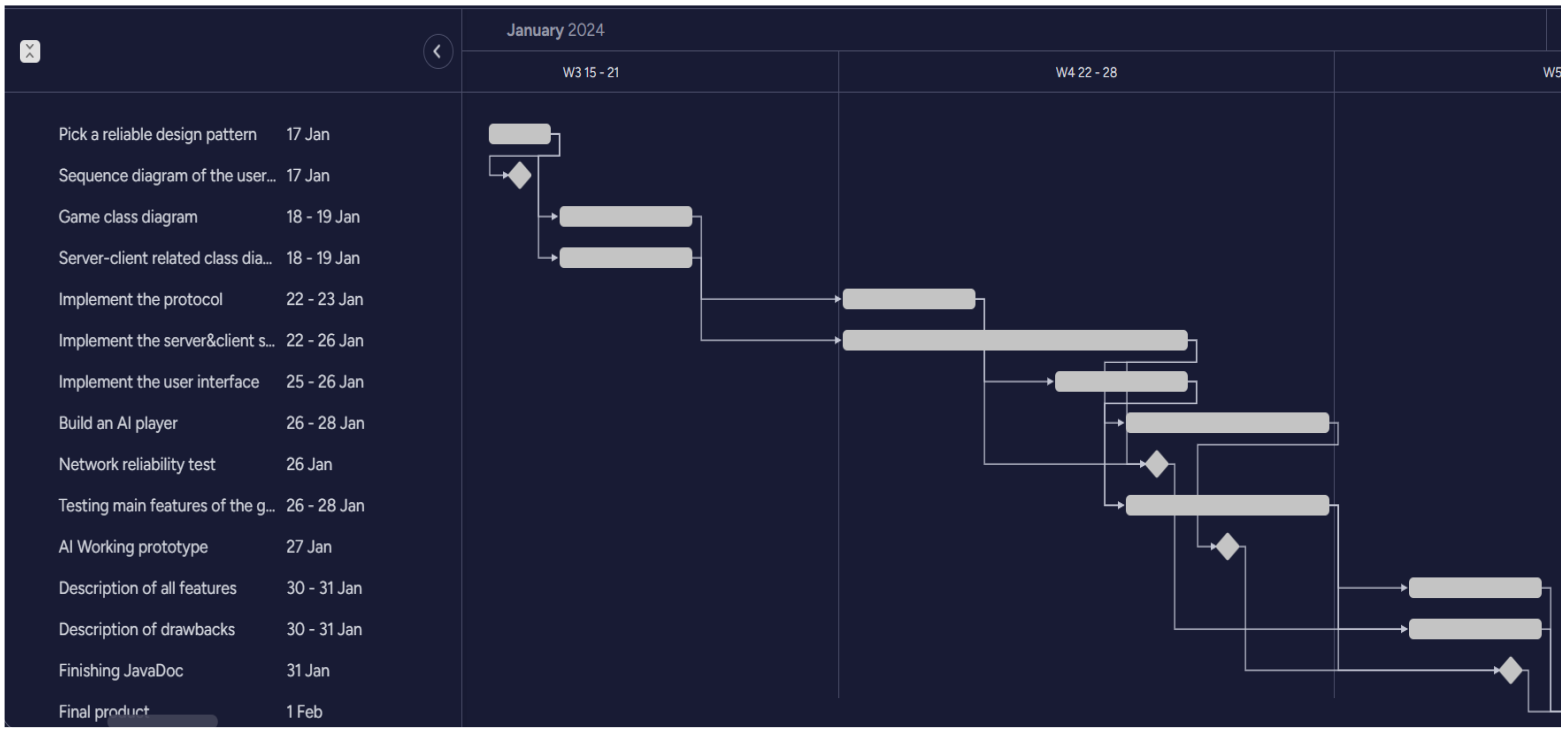
Note: unit tests will be written in parallel with the code

WP3 Title: ***Testing & Debugging***
Timeline: - Start Date: 26 Jan 2024 - End Date: 29 Jan 2024
Description:
Testing and Debugging package will be done after the general implementation. In this part, different tests will be written in order to make the system more reliable and thread-safe.
Tasks:
Task 1: Testing main features of the game
- Subtask 1.1: test game logic - Subtask 1.2: test game behavior - Subtask 1.3: test the user-friendliness of the user inerface
Task 2: Test AI potential
Task 3: Testing Network reliability - Subtask 3.1: test the system performance under the normal load - Subtask 3.2: test the system performance under the pick load
Dependencies:
T1.1 and T1.2 depends on WP2-T1, T1.3 on WP2-T3,T2 depends on WP2-T4, T3 depends on WP2-T3 **Resources:** - Human Resources: - Danil Badarev: Task 2; Subtask 1.1; Subtask 1.2 - Mira Gozbenko: Subtask 1.3, Task 3 - Material Resources: - IntelliJ IDEA 2023.2.5, https://github.com, Visual Studio, Microsoft Office
Risks and mitigation strategies:
User interface testing complexity: involve other people in user interface testing for getting an independent opinion
Time constraints on testing AI: use parallel testing for time optimization
Collaboration: strong pair programming, version control, online meetings - 30%, person to person meetings – 70%, total 22 hours spent. We won't work on Sunday and during the lunch time.

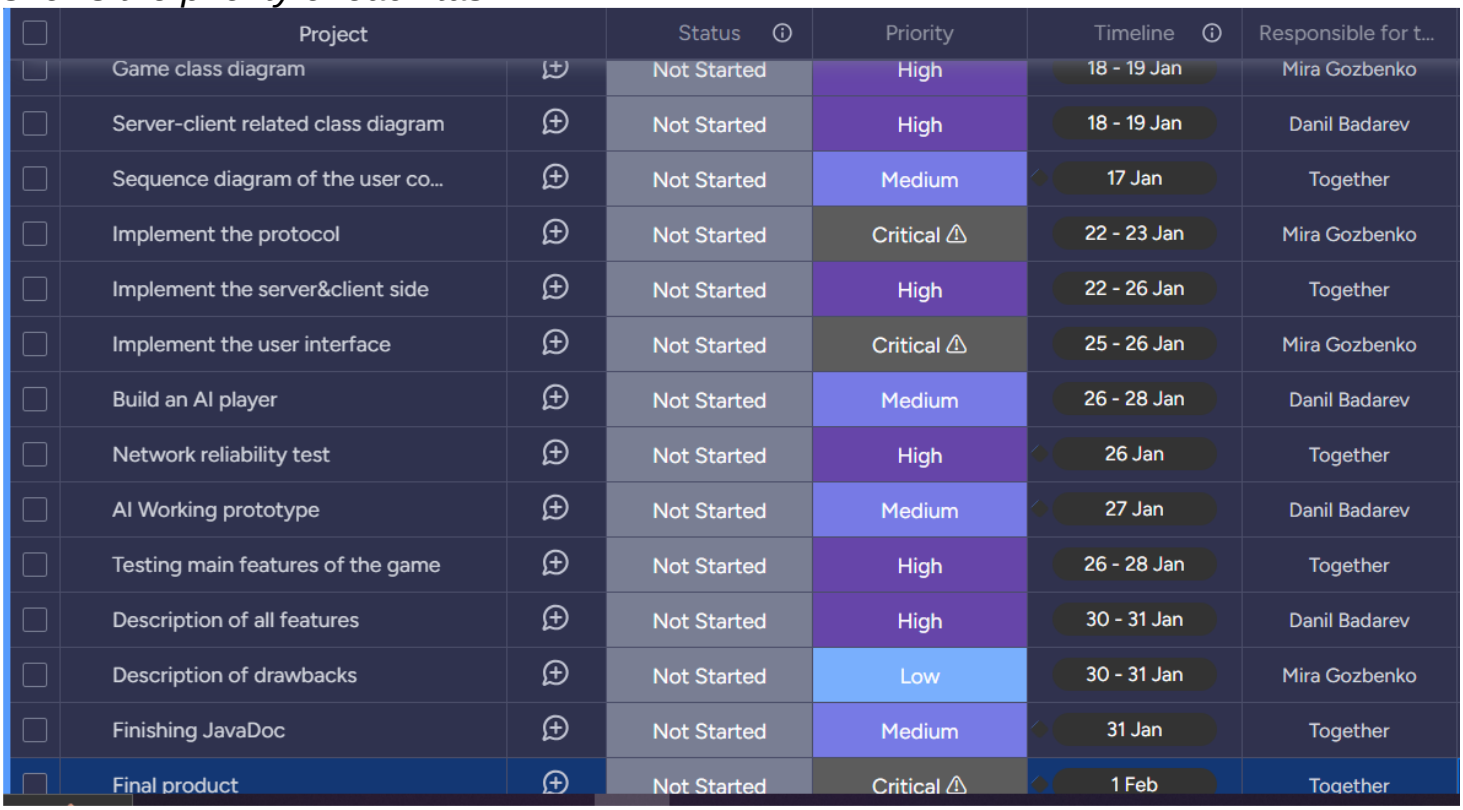
Progress Tracking:
Milestones: 1 – The midway submission (28 Jan), 2 – User feedback (26 Jan), 3 – AI Working prototype (27 Jan), 4 – Network reliability test (26 Jan)
Recognizing Achievements: Every milestone will be kept in mind and support a positive mindset of the developers.
Notes: A week meeting should be conducted in order to avoid any missunderstanding
WP4 Title: ***Final report and documentation***
Timeline:
- Start Date: 30 Jan 2024 - End Date: 1 Feb 2024
Description: The Final Report and Documentation work package summarizes and reveals the outcomes of the project. It involves tasks such as writing formal specifications, documentation, and deploying the finalized project.
Tasks: Task 1: Write the JML - Subtask 1.1: write preconditions - Subtask 1.2: write postconditions Task 2: Write JavaDoc Task 3: Create the final report
- Subtask 3.1: write the description of all features - Subtask 3.2: compare the requirements with the completed features - Subtask 3.3: compare the expectations with the final system - Subtask 3.4: describe drawbacks of the system
Task 4: Project deploying
Dependencies:
T1 depends on WP2. To start working on Task 3&4, Task 1& 2 should be finished.
Resources: Human Resources: Danil Badarev: Task 1; Task 2; Task 4
Mira Gozbenko: Task 3; Task 4 **Material Resources:** - IntelliJ IDEA 2023.2.5, https://github.com, Microsoft Word, Excel

Risks and mitigation strategies:
Lack of collaboration on Task 2: Combine pair programming and individual work with effective version control. Regular reports during daily meetings to ensure everyone is on the same page.
Delays in Task 3 due to subtask quantity: Allocate time effectively, dedicating 40% of the time to Subtask 3.1 and 45% to Subtasks 3.2 & 3.3. Reserve 15% for Subtask 3.4. Use online communication via Discord if additional collaboration is needed.
Collaboration: Daily stand-up meetings for progress updates. Task 2 will be team effort with strong pair programming and individual work, with version control using GitLab. We will have daily stand-up meetings. Task 3 will be executed in parallel, dedicating 40% of the time to Subtask 3.1 and 45% to Subtask 3.2&3.3 and 15% to Subtask 3.4, with online communication via Discord if needed. We will only work on weekends if it's necessary.
Progress Tracking: Milestones: 1 – Writing description for all functions (29 Jan), 2 – Finishing JavaDoc (31 Jan), 3 – Writing recommendations for the future updates (31 Jan), 4 – final product (1 Feb) **Note**: The parts of the final project will be split between the members. However, in the end of the timeline both members have to check the final version of the report.
Recognizing Achievements: Every milestone will be kept in mind and support a positive mindset of the developers.

(T5)Gantt Chart



The following table describes who is working on what task. Moreover, it shows the priority of each task:



Project	Status	Priority	Timeline	Responsible for t...
Game class diagram	Not Started	High	18 - 19 Jan	Mira Gozbenko
Server-client related class diagram	Not Started	High	18 - 19 Jan	Danil Badarev
Sequence diagram of the user co...	Not Started	Medium	17 Jan	Together
Implement the protocol	Not Started	Critical ⚠	22 - 23 Jan	Mira Gozbenko
Implement the server&client side	Not Started	High	22 - 26 Jan	Together
Implement the user interface	Not Started	Critical ⚠	25 - 26 Jan	Mira Gozbenko
Build an AI player	Not Started	Medium	26 - 28 Jan	Danil Badarev
Network reliability test	Not Started	High	26 Jan	Together
AI Working prototype	Not Started	Medium	27 Jan	Danil Badarev
Testing main features of the game	Not Started	High	26 - 28 Jan	Together
Description of all features	Not Started	High	30 - 31 Jan	Danil Badarev
Description of drawbacks	Not Started	Low	30 - 31 Jan	Mira Gozbenko
Finishing JavaDoc	Not Started	Medium	31 Jan	Together
Final product	Not Started	Critical ⚠	1 Feb	Together

List of Danil's Tasks:

1. Watch "Project reveal and kickoff"
2. Make an apointment with the psyhologist
3. Prepare for the Programing test
4. Write this document for Career Skills
5. Plan approaching holidays
6. Fill the poll (close deadline)
7. Start wathcing a new series
8. Make a plan for approaching the programming project
9. Finish the chat server program from the week 7
10. Watch some java lectures to improve my programming skills
11. Get my bike repaired
12. Get my room cleaned
13. Read the book called "Fluent python" (work related)
14. Prepare questions for programming lectorial
15. Sign up for the next module

NECESSITY: 4, 9, 11 QUALITY: 1, 2, 3, 5, 8, 10, 13, 14, 15 DECEPTION: 6, 12 WASTE OF TIME: 7

List of Mira's tasks:

1. Create a class diagram for the project
2. Fill out the programming sample test
3. Clean my PC

4. Design an AI algorithm
5. Create a sketch of the user interface
6. Take out the trash
7. Write testing reports
8. Re-watch programming videos
9. Play Minecraft
10. Sign up for the next module
11. Go to design exam review
12. Write my resume
13. Look for internships
14. Work on side project
15. Attend honor's interview

NECESSITY: 2,3,8,11,15 QUALITY:1,4,5,7,10,12,13,14 DECEPTION: 6 WASTE OF TIME: 9

Reflection:

The tasks align with the WPs and Gantt chart from program design to testing. The workload is evenly distributed, with tasks supporting different project phases, which ensures balanced progress. The prioritization of tasks is reasonable, focusing on activities important to the general success. There are no visible conflicts within the Eisenhower matrices

Identification of one success and one shortcoming

Success: Building the game logic without a client and server and testing it. The game part of the project seems to be totally complete.

Shortcoming: The delay in building the networking related part due to the poor time management.

General task division

The initial task division was revised and changed to the following:

Mira Gozbenko: Report writing (except the parts related to testing), writing the part of the game related to game logic, writing the Server and part of the client

Danil Badarev: Writing part of the game logic, documenting the code and cleaning, writing AI, connecting the game, server and client, testing, writing the report part related to testing

Specific collaboration styles used overview with justification of usage

As we mentioned earlier, we used the **strong pair programming** to overcome the difficult code parts, which insured we both understand it and improved our coding and cooperation skills. The second collaboration style used was **daily stand-up meetings**, where we discussed the work done separately and approached the difficulties appeared, which was an essential tool for keeping track of the overall progress and addressing the delays.

Methods used to ensure that all group members were familiar with all parts of the code

As mentioned in the previous section, we used **daily stand-up meetings** to ensure we are aware of the code the other person wrote. During the daily meetings we presented the work done to each other and explained the code we wrote. Additionally, during the meetings we sent the written code so the other person had an opportunity to review it when convenient. If questions arose afterwards, **communication through discord** was used, either video-calls or texting, to explain the part that created confusion.

The lessons learnt from the success and the shortcoming

From the success: Good initial project planning, collaboration, task division and putting away backup time was essential for the finishing of the game in time. The same process should be repeated in upcoming projects.

From the shortcoming: The plan didn't consider the difficulty of the server and client creation because of lack of experience in building those parts. In the future, the time needed for similar parts of projects will be taken into consideration.