

**MOD2 Programming Project
Resit-13**

Filip Ionas – s2960249

Narendra Setty – s2944200

Index

Introduction	2
Reflection on Initial Design	3
Game Logic:.....	3
Server-Client.....	3
Design explained	4
Package: AI	4
Package: client.....	5
Package: game.....	5
Package: server	6
Package: test	7
Concurrency Mechanism	8
Client Side:	8
Server Side:	8
Testing & Test Coverage	9
System Tests	11
Reflection on the process.....	16
Filip.....	16
Narendra	17

Introduction

Dots and boxes is a board-game in which 2 players take turns placing lines one at a time with the aim of completing more boxes than your opponent. The project for this year's Software Systems Module was to implement this game into a digital environment with the use of Java and the skills picked up along the course of these 10 weeks. This report will provide insight into our process, and it will explain how our implementation functions and the design choices we have made.

The scope of our project was to create a functioning system that will allow you to play dots and boxes online with any opponent, but also allow you to play against computers. The project would be composed of the main game logic and the client/server architecture that would support online play.

In the following sections we will explain the design behind our implementation, and we will give explanations for the approach we chose to take.

Initially we talk about the initial design that we created and the alterations that we made for the final submission. Then we talk about the design of our project in more detail with descriptions of classes and packages and how they interact with each other as well as the use of techniques like abstraction and modularization. Following this we talk about the concurrency mechanism of the program focusing on both the server and the client. For the next part we explain our testing strategy, our usage of test coverage as well as the different system tests used to test our final implementation.

Reflection on Initial Design

Game Logic:

Initially we only had the basic structure consisting of different classes that could be used. We hadn't realised the various methods and fields within those classes. One of the major changes made in the game logic regarding the different classes was the addition of abstract class `AbstractPlayer` to allow for abstraction. This means that different types of players can extend the basic functionalities of the `AbstractPlayer` class and play the game.

Server-Client

Client: Our initial design of the client was pretty simple, the biggest addition we have made is a different class for an AI client, and a client connection class, the client connection represents the connection between the client and the server. Two main missing elements in our final design are the GUI which was a bonus we thought we would have time for, but module 6 ended up getting in the way and the chat listener which we believed would be easy to implement and then realised we misunderstood what the requirements were. Otherwise our design stayed very similar, keeping the modularity for the UI and Listener, a feature which allows us to modify the UI without modifying any of the other classes.

Server: For the design of the server we initially had a very basic understanding of how it would work and hence we only thought of three classes. For the final design we added the abstract classes `SocketServer` and `SocketConnection` in order to accept connection from clients and to handle the different types of messages respectively. These allow for abstraction so that many other different types of classes can be extended from those two classes and only the desired functionalities will be known by the user and not the inner workings. We also added the `Protocol` class that consists of the fields representing the different protocol messages used for server-client communication. Additionally, `ServerState` enum was also added representing the current state of the server-client connection to simplify the handling of messages. Finally we also added the `GameServer` class that represents the game being played by two clients from the queue in the server. It contains an instance of the `DotsAndBoxesGame` class that allows it to modify the game state in the manner it is needed.

Design explained

Use of different methods for Object Oriented Design:

Abstraction: It is a concept used to simplify complex systems by representing essential features without showing unnecessary details. It can be implemented by the use of abstract classes that define a set of common methods and properties shared by multiple classes. For example, in the client side we have used UI class as an abstract class representing the user interface. It consists of a set of common methods and fields used by different types of user interfaces. The two different user interfaces that extend UI are TUI and AIUI representing the user interfaces for human and automated players respectively. The UI class made it easier to create different types of user interfaces for the client.

Encapsulation: Private Fields within classes are used to restrict direct access from other classes. In each class all the essential fields are set to private and when required there are public getter and setter methods to allow other classes to access them. This helps in hiding the internal implementation details, providing controlled access to the class's state, and facilitating better code organization and maintenance. For example in the DotsAndBoxesGame class there are private fields for player1Score and player2Score which can be accessed via getter methods in the same class.

Modularization: Involves breaking down the program into packages each with their own desired functionality. For example in our project we have different packages for the game, client, server ai and tests. They are structured in such a way that if there is a problem in one of the packages the functionalities of the other packages will not be affected.

Package: AI

Explanation: In this package we have our AI objects, we have separated it into 3 classes representing different levels of difficulty, and all the files in this package rely on game.AbstractPlayer.

Classes	Description
Dumb AI	An AI that randomly generates a valid move
Normal AI	An AI that tries to close boxes, if no closing moves are available, it will randomly choose a valid move.
Smart AI	An AI that tries to perform moves that will not give the opponent the opportunity to close a box, but also closes boxes when available.

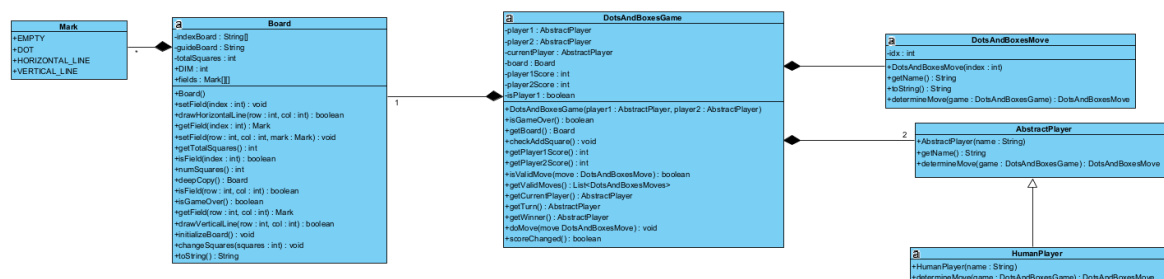
Package: client

Explanation: In this package we have the files necessary in order to run the client, it contains a generic UI class which allows for different types of UI to be implemented without any changes needed to the client. It also has an abstract listener in case the messages incoming from the server need to be handled in a different way.

Classes	Description
AIUI	A user interface which creates a bot player that can connect to any Dots and Boxes server.
Client	The core class of the client, an object of this kind is created when the UI is started and it handles games, outgoing messages and other commands.
ClientConnection	This is the connection between the client and server, it also creates listeners that will handle incoming messages.
ClientProtocol	A class used for formatting messages and handling appropriate actions such as creating new games, giving a list of valid moves and much more.
GameListener	Listens for incoming messages from the server and handles them using ClientProtocol
Listener	An abstract class meant for the creation of different types of listeners.
TUI	A textual user interface, that a user can use in order to join a game of Dots and Boxes.
UI	An abstract class used for the creation of different types of UI.

Package: game

Explanation: In this package the actual game of the project is presents. It consists of the board where the game is played and the DotsAndBoxesGame class where the game logic is handled. There is also AbstractPlayer class that represents the player that represents the player in the game.

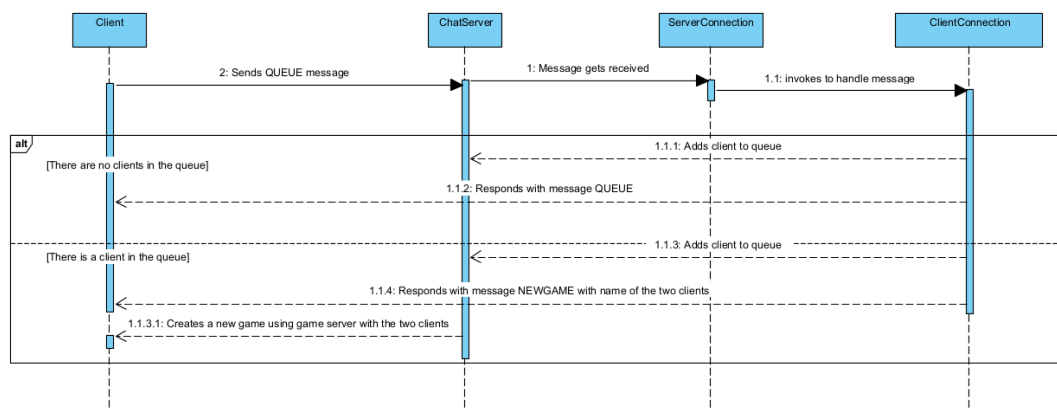
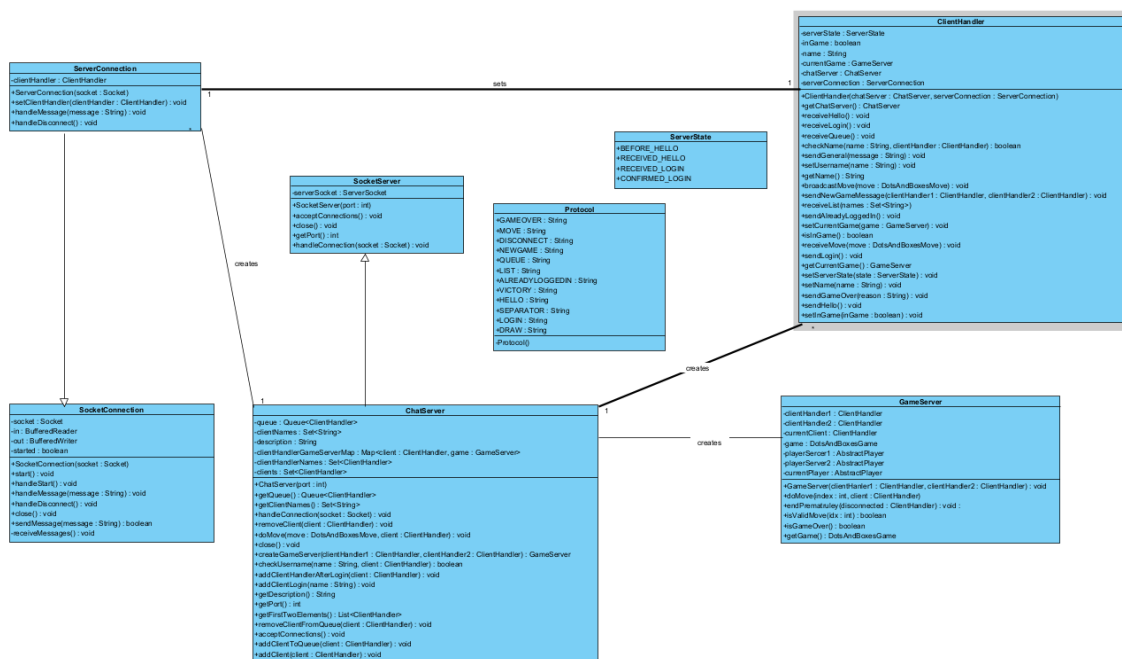


Classes	Description
---------	-------------

AbstractPlayer	The abstract class representing the player in the game
Board	This class is the board of the Dots and Boxes game
DotsAndBoxesGame	Implements the game logic of Dots and Boxes
DotsAndBoxesMove	Encapsulates a move using an index in the game
DotsAndBoxesTUI	
HumanPlayer	Enables the user to make the moves in the game and extends AbstractPlayer

Package: server

Explanation: This package contains the classes that handles the server-side code of the connection. It consists of classes such as ChatServer which is the main server class for accepting connections from clients and also has classes like ClientHandler and ServerConnection that receive and send messages back to the client based on the state of the server and the message sent by the client.



Above we have given an example of a sequence diagram of clients joining a queue. If a client joins a queue, the `ServerConnection` class is invoked which in turn invokes the appropriate method in `ClientHandler` which in turn adds the client to the queue using the instance of `ChatServer` present in the `ClientHandler` class. And depending on whether the queue is empty or not different messages are sent back to the client. If the queue is empty a simple 'QUEUE' message is sent bac to the client but if the queue is not empty a new game is created consisting of the two clients and a 'NEWGAME' message is sent to both the clients.

Classes	Description
ChatServer	Represents the server in Dots and Boxes game that accepts connections and handles the different client appropriately. It extends <code>SocketServer</code> .
ClientHandler	Represents a handler for client interactions with the server. It manages client states, processes client actions such as receiving messages and making moves in a game, and facilitates communication between clients and the chat server.
GameServer	Contains information about the game between two clients.
Protocol	Contains the different protocol types used in messages between server and client.
ServerConnection	Handles the different types of messages the server receives from the client. It extends <code>SocketConnection</code> .
ServerState	Enum that represents the different states at which the client-server session is in.
SocketConnection	Abstract class that reads single line messages from the socket.
SocketServer	Abstract class that represents the server that accepts connections from clients.

Package: test

Explanation: This Package contains all the JUnit tests we have created. If a feature does not have a test present in this files then it has been tested manually.

Classes	Description
AITest	Runs games between different AI in order to make sure that games get finished and a winner is assigned correctly.
ClientTest	Tests the initialization of the client, and any other feature that is possible to do before joining a new game.
GameTest	Tests the behaviour of our game logic.

Concurrency Mechanism

Concurrency refers to the synchronization of multiple tasks or threads in order to make them run seamlessly and simultaneously. In our project this plays a large role in our implementation of a client and server, and that is where all of the concurrent threads can be found.

Client Side:

To start with, we have our client which generates a new thread for every “listener” attached to the client, our current implementation only makes use of one but it has the ability to be expended. This listener thread, waits for incoming messages from the server, the name listener deriving from the fact that it “listens” to the server connection, and then handles incoming data. The main thread handles the user input, handling outgoing messages. These two threads share multiple objects, the client, the client connection and most importantly the client protocol. The client protocol being a shared object is necessary because both incoming and outgoing messages need to be formatted following the protocol, and while we’re modifying the messages to fit the protocol, we can take care of some logistical problems, in this class we also determine whether a player is in the queue or in a game. Fortunately no concurrency issue presents itself from these two Booleans since the nature of the server, the speed at which a user can input commands and the game itself stop the user from entering and ending a new game simultaneously, and the same happens with queue. One improvement for the synchronization of the client would be the way we display the information, right now not every command waits for a response from the server before allowing you to input a new command, we have solved this problem for most commands, but others have just slipped by us, fortunately it cannot result in any issues, it just means that sometimes the client doesn’t look as pretty as we would have liked it to.

Server Side:

On the server side synchronisation plays a more important role. First of all our main class will listen for new connections and create a new thread for every client that joins, this thread will listen to incoming messages and take appropriate actions for every incoming command. The main shared object on the server side is an instance of ChatServer which then handles any action needed such as adding people to queue, creating games and any other game activity that may be needed. Here concurrency issues start to appear, one of the main synchronization problems we had to solve was adding and removing people from the queue. Initially the problem we were having was that if three people were in the queue, two games would be created by two different threads, but by the time the second thread got the people from the queue to create a game object, the first thread would remove two of the players from the queue, leaving the second thread with only one player and resulting in an error. Other than that synchronization is needed since multiple threads have access to shared lists, which if not handled would result in concurrent modification errors.

Testing & Test Coverage

Test coverage metrics are a tool used for measuring the level at which aspects of the code have been tested. They provide insight into the quality of the existing tests, and they give you more insight into methods and features that still require testing. They are usually represented by a percentage which represents the amount of lines, methods or classes that have been successfully tested, divided by the total amount existing in the project or files covered by the tests.

We will mainly be making use of the following types of coverage:





Firstly we believe that assuring that the methods we have implemented function accordingly is one of the most important parts of the project, thus we will make use of method coverage in order to guarantee that as many of our methods act as expected as possible. This metric measures the amount of methods that have been covered by the tests and depending on the tool used it can show us which methods have not been covered by the testing yet.

Another useful metric is class coverage, this measures the amount of classes covered by the testing, and it can give us insight into the interoperability of our classes. We aim to make use of this in order to assure that classes work together nicely.

Lastly we will be making use of line coverage, since it's another of the metrics given in the tool of our choice IntelliJ, this measures the amount of lines covered by our test, and it can come in useful in order to weed out edge cases and other non-obvious bugs that could be present.

These metrics will help us guarantee that our game and game logic work not only accordingly to the original rules of Dots and Boxes but also function as a proper program with minimal bugs. Our hope is to obtain a code coverage of at least 95%, and in order to achieve this we have kept up to date with the tests, creating a new one alongside each big feature implemented.

Our current testing coverage is

>  src.client	80% (8/10)	43% (25/58)	35% (92/259)
>  src.Server	90% (9/10)	64% (46/71)	54% (146/26...
>  src.game	87% (7/8)	86% (39/45)	76% (224/29...
>  src.AI	100% (3/3)	100% (9/9)	98% (60/61)

We will break it down package by package in order to make the explanation clearer. We will start with the client, if we take a look at the coverage of each class we have the following:

src.client	60% (6/10)	37% (22/58)	32% (83/259)
UI	100% (1/1)	0% (0/1)	50% (1/2)
TUI	0% (0/1)	0% (0/10)	0% (0/62)
AIUI	0% (0/1)	0% (0/12)	0% (0/50)
Client	33% (1/3)	33% (5/15)	36% (14/38)
GameListener	100% (1/1)	75% (3/4)	66% (6/9)
ClientProtocol	100% (1/1)	83% (10/12)	56% (42/74)
Listener	100% (1/1)	100% (0/0)	100% (1/1)
ClientConnection	100% (1/1)	100% (4/4)	82% (19/23)

Here the classes UI, TUI and AIUI are not tested using JUnit since they represent a user interface, something we decided to test manually in order to guarantee that the user experience is as good as we could make it. The functionality of the client is mostly tested in the file ClientTest, here we cover most of the features of the client, but due to some synchronization problems we couldn't write tests for any event that happens after the message NEWGAME, these parts have been thoroughly manually tested, so we did not feel the need to solve the synchronization issues in the tests.

Next we have the AI classes, these have been fully covered, but their tests do not contain any asserts, the test print out the game between two AI of different levels of intelligence. This is the best way of accomplishing the tests since due to the random moves chosen by the AI we could not determine the outcome of a game and as such no asserts could be written. Instead we went over the games manually and verified that the AI acts as planned, and that all the moves are performed correctly and legally.

src.Server	90% (9/10)	64% (46/71)	54% (146/26...
PlayerServer	0% (0/1)	0% (0/2)	0% (0/3)
GameServer	100% (3/3)	37% (3/8)	19% (10/52)
SocketServer	100% (1/1)	50% (2/4)	54% (6/11)
SocketConnection	100% (1/1)	62% (5/8)	56% (17/30)
ChatServer	100% (1/1)	70% (14/20)	55% (49/88)
ClientHandler	100% (1/1)	73% (17/23)	79% (38/48)
ServerConnection	100% (1/1)	75% (3/4)	70% (24/34)
Protocol	100% (0/0)	100% (0/0)	100% (0/0)
ServerState	100% (1/1)	100% (2/2)	100% (2/2)

The server classes have mostly been tested using the client tests, since those tests gave positive results it means that the server has handled the commands properly and returned

the expected message, the behaviours of starting new games and playing games have all been tested manually due to the reasons mentioned above.

▼ src.game	87% (7/8)	86% (39/45)	76% (224/29...
🟢 DotsAndBoxesTUI	0% (0/1)	0% (0/3)	0% (0/39)
🟢 HumanPlayer	100% (1/1)	50% (1/2)	5% (1/18)
🟢 DotsAndBoxesGame	100% (1/1)	84% (11/13)	88% (47/53)
🟢 DotsAndBoxesMove	100% (1/1)	100% (3/3)	100% (4/4)
🟢 AbstractPlayer	100% (1/1)	100% (4/4)	100% (5/5)
🟢 Board	100% (3/3)	100% (20/20)	96% (167/17...

And finally the game package, these classes have been almost completely covered, we have added the appropriate tests that were mentioned during the feedback session. These have been useful since we have discovered a couple of edge cases from these tests and we were able to solve them. The TUI has not been tested for the same reasons the previous TUI was not tested, while human player has not been fully tested since the determineMove method requires human input from the console, this has been tested manually and we are certain of it's functionality.

Overall our coverage is not perfect but we are satisfied with the tests we have written, and they have proved useful during the implementation process.

System Tests

Testing Report for Functional Requirement 1:

Functional Requirement 1: When the server is started, it will ask the user to input a port number where it will accept connections on. If this number is already in use, the server will ask again.

Expected behaviour: The user starts a server and enters a valid port number and there is a message indicating that the server has been started at that port.

Testing Result:

1. User: Starts server
2. System: Enter a port number
3. User: Enter 4567 as server port
4. System : Server started at port 4567

Alternative case: In step 3 user enters invalid port number like -1. System response will be: Invalid port. Enter a port number

Testing Report for Functional Requirement 2:

Functional Requirement 2: When the client is started, it should ask the user for the IP-address and port number of the server to connect to.

Expected behaviour: After a server is started the user also starts a client. Then the user is asked for a server address and a port number. After the user fills in the appropriate details there is a message the client receives indicating that the client is connected to the server.

Testing Result:

1. User: Starts client
2. System: Input Server Address (Leave empty for LocalHost):
3. User: Leaves empty and click enter
4. System: Input Server Port (Leave empty for 4567):
5. User: Leaves empty and click enter
6. System: HELLO~Client by Somone
Input Username:
HELLO~DotsAndBoxesServer

Testing Report for Functional Requirement 3:

Functional Requirement 3: The server can support multiple clients

Expected Behaviour: When a server is started on port number (say 4567) multiple clients can connect to it using its address and the relevant port number without any errors occurring.

Testing Result:

1. User: Starts server
2. System: Enter a port number
3. User: Enter 4567 as server port
4. System : Server started at port 4567
5. User: Starts client
6. System: Input Server Address (Leave empty for LocalHost):
7. User: Leaves empty and click enter
8. System: Input Server Port (Leave empty for 4567):
9. User: Leaves empty and click enter
10. System: HELLO~Client by Somone
Input Username:
HELLO~DotsAndBoxesServer
11. User: Enter name 'Bro'
12. System: LOGIN
13. User: Starts client
14. System: Input Server Address (Leave empty for LocalHost):
15. User: Leaves empty and click enter
16. System: Input Server Port (Leave empty for 4567):
17. User: Leaves empty and click enter
18. System: HELLO~Client by Somone
Input Username:

HELLO~DotsAndBoxesServer

19. User: Enter name 'Dude'
20. System: LOGIN
21. User: LIST~Dude~Bro

Testing Report for Functional Requirement 4:

Functional Requirement 4: The server only allows clients with unique usernames to login

Expected Behaviour: When a client is logged in with a username and another tries to login with the same username the second client receives a message indicating that the username already exists and they should choose a different name.

Testing (Prerequisite: Server started at port 4567):

1. User: Starts a client
2. System: Input Server Address (Leave empty for LocalHost):
3. User: Leaves empty and click enter
4. System: Input Server Port (Leave empty for 4567):
5. User: Leaves empty and click enter
6. System: HELLO~Client by Somone
Input Username:
HELLO~DotsAndBoxesServer
7. User: Enter name 'Player'
8. System: LOGIN
9. User: Starts a client
10. System: Input Server Address (Leave empty for LocalHost):
11. User: Leaves empty and click enter
12. System: Input Server Port (Leave empty for 4567):
13. User: Leaves empty and click enter
14. System: HELLO~Client by Somone
Input Username:
HELLO~DotsAndBoxesServer
15. User: Enter name 'Player'
16. System: ALREADYLOGGEDIN
Input Username:

Testing Report for Functional Requirement 5:

Functional Requirement 5: When two players enter a queue a new game will be created consisting of those two players

Expected behaviour: When two clients are logged in to a server and they enter a queue using the QUEUE command one after the other a new game will be created consisting of those clients as the two players.

Testing (Prerequisite: Two clients (for example with names player1 and player2) are logged in):

1. User: As player1, write command QUEUE

2. System: QUEUE
3. User: As player2, write command QUEUE
4. System:

```

QUEUE
Player 1 score: 0
Player 2 score: 0
*   *   *   *   *   *   *   00 *   01 *   02 *   03 *   04 *
05   06   07   08   09   10
*   *   *   *   *   *   *   11 *   12 *   13 *   14 *   15 *
16   17   18   19   20   21
*   *   *   *   *   *   *   22 *   23 *   24 *   25 *   26 *
27   28   29   30   31   32
*   *   *   *   *   *   *   33 *   34 *   35 *   36 *   37 *
38   39   40   41   42   43
*   *   *   *   *   *   *   44 *   45 *   46 *   47 *   48 *
49   50   51   52   53   54
*   *   *   *   *   *   *   55 *   56 *   57 *   58 *   59 *

It's player1's Turn
NEWGAME~player1~player2

```

Testing Report for Functional Requirement 6:

Functional Requirement 6: When two clients are in a game and one of the clients disconnect, the server shouldn't crash and should send the appropriate game over message.

Testing (Prerequisite: Two clients (for example with names player1 and player2) have started a game):

1. User: As player1, disconnect the player by stopping the running client
2. System: GAMEOVER~DISCONNECT~player2

Additional information: After the above steps the server did not crash and other clients were still able to connect to it.

Testing Report for Functional Requirement 7:

Functional Requirement 7: When the client is controlled by a human player, the user can request a possible valid move as a hint via the TUI

Expected Behaviour: After two clients are in a game started by the server and one of the human clients types a command to ask for a list of possible moves they get a response consisting of those moves.

Testing (Prerequisite: The server has started a game consisting of at least one human player):

1. User: As first player writes 'move 5'

2. System:

```

move 5
MOVE~5
MOVE~5
Player 1 score: 0
Player 2 score: 0
*      *      *      *      *      *      *      00  *      01  *      02  *      03  *      04  *
|      |      |      |      |      |      |      05      06      07      08      09      10
*      *      *      *      *      *      *      11  *      12  *      13  *      14  *      15  *
*      *      *      *      *      *      *      16      17      18      19      20      21
*      *      *      *      *      *      *      22  *      23  *      24  *      25  *      26  *
*      *      *      *      *      *      *      27      28      29      30      31      32
*      *      *      *      *      *      *      33  *      34  *      35  *      36  *      37  *
*      *      *      *      *      *      *      38      39      40      41      42      43
*      *      *      *      *      *      *      44  *      45  *      46  *      47  *      48  *
*      *      *      *      *      *      *      49      50      51      52      53      54
*      *      *      *      *      *      *      55  *      56  *      57  *      58  *      59  *

```

3. User: As player 2 writes 'HINT' to get the list of possible moves
4. System:

```

HINT
VALIDMOVES~0-1-2-3-4-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32-33-34-35-36-37-38-39-40-41-42-43-44-45-46-47-48-49-50-51-52-53-54-55-56-57-58-59

```

Testing Report for Functional Requirement 8:

Functional Requirement 7: When a human plays a game with the AI, the AI can perform the moves on the board without the need for human intervention for the entire game

Expected behaviour: When two clients one human and one AI are in a game started by the server, when the human makes a move, the AI client immediately responds with a move of its own following which the human can play again and so on.

Testing (Prerequisite: The server has been started at port 4567 and a human client player1 has logged in):

1. User: Starts the AI TUI
2. System: Input Server Address (Leave empty for LocalHost):
3. User: Leaves empty and click enter
4. System: Input Server Port (Leave empty for 4567):
5. User: Leaves empty and clicks enter
6. System: HELLO~Client by Somone
HELLO~DotsAndBoxesServer
Choose Difficulty: easy/normal/hard
7. User: choose normal
8. System: LOGIN~#DotsAndBoxesNormalBot5988
QUEUE
LOGIN

9. User: As player1 join the queue using 'QUEUE'

10. System:

```
* * * * * * 00 * 01 * 02 * 03 * 04 *
05 06 07 08 09 10
* * * * * * 11 * 12 * 13 * 14 * 15 *
16 17 18 19 20 21
* * * * * * 22 * 23 * 24 * 25 * 26 *
27 28 29 30 31 32
* * * * * * 33 * 34 * 35 * 36 * 37 *
38 39 40 41 42 43
* * * * * * 44 * 45 * 46 * 47 * 48 *
49 50 51 52 53 54
* * * * * * 55 * 56 * 57 * 58 * 59 *

It's #DotsAndBoxesNormalBot5988's Turn
NEWGAME~#DotsAndBoxesNormalBot5988~player1
MOVE~46
Player 1 score: 0
Player 2 score: 0
* * * * * * 00 * 01 * 02 * 03 * 04 *
05 06 07 08 09 10
* * * * * * 11 * 12 * 13 * 14 * 15 *
16 17 18 19 20 21
* * * * * * 22 * 23 * 24 * 25 * 26 *
27 28 29 30 31 32
* * * * * * 33 * 34 * 35 * 36 * 37 *
38 39 40 41 42 43
* * * --- * * 44 * 45 * 46 * 47 * 48 *
49 50 51 52 53 54
* * * * * * 55 * 56 * 57 * 58 * 59 *

It's player1's Turn
```

11. User: As player1 write 'move 22'

Reflection on the process

After all the weeks we spent working on the project, we thought it appropriate to add some final thoughts on the process as a whole. We have chosen to separate this in two parts, one for each member of the team.

Filip

This being the second time I have taken part in the project, the pressure and challenges present last year were not as oppressing, even though my time was split between module 6 and this module. At times it felt like I needed to be in two places at once, but luckily my teammate was understanding and we managed to see eye to eye on the progress of this project. The main bulk of work started

after the winter break, by that time my partner managed to complete most of the game logic and the only thing left to do on that part was for me to help solve some of the bugs, and help designing the score system. The rest of the work I have done on the project was to implement the AI, Client and Tests, in order to make up for the little work I have done on the game logic. Throughout the process even though we split the work, we often had meetings, discussed our progress and ask each other for help when needed. Unfortunately we didn't manage to add all the bonus features we wanted to add, but on my part studying for module 6 came in the way, so I am happy with the final product we managed to deliver, and compared to last year I can only see this project as a positive experience.

Narendra

Initially we did not make a concrete plan for the project. From previous experience we knew that the amount of work would be large. This was the reason I started earlier with the game logic during the vacation as I wanted to leave enough time to complete the tasks. Filip was of great help in the networking part of the project not just with the client part but also with the server part which I was supposed to implement. Whenever I had a doubt, he would help me figure out the answer. Due to the fact that we had an early start with the project we were able to divide the tasks in a more effective and realistic manner. After Filip had completed the tests we both contributed to different sections of the report cross checking each other's work. We met either at university or on discord whenever one of us had a doubt and we had regular meetings at least once in every two days. We were also willing to explain our code to each other in case one of us had questions. However we had planned to complete at least one of the bonus tasks however due to the workload of both module 2 and parts of module 6 we were unfortunately unable to complete this goal. From this experience we learned that starting the work as early as possible was crucial to completing the tasks efficiently.