A Mini-Project Report

on

“A Multiplayer Chess Game”

Submitted to the
Pune Institute of Computer Technology, Pune In
partial fulfillment for the award of the Degree of
Bachelor of Engineering
in
Information Technology
by

Rohit pendse	43356
Harshal Rajput	43361
Nisha Rudrwar	43364
Bhavana Sanap	43365
Satyajeet Sankpal	43366

Under the guidance of

Prof. J. K. Kamble

Department Of Information Technology
Pune Institute of Computer Technology College of Engineering
Sr. No 27, Pune-Satara Road, Dhankawadi, Pune - 411 043.

2022-2023

CERTIFICATE

This is to certify that the project report entitled

A MULTIPLAYER CHESS GAME

Submitted by

Rohit pendse	43356
Harshal Rajput	43361
Nisha Rudrwar	43364
Bhavana Sanap	43365
Satyajeet Sankpal	43366

is a bonafide work carried out by them under the supervision of **Prof. J. K. Kamble** and it is approved for the partial fulfillment of the requirement of **Lab Practice-V** for the award of the Degree ofBachelor of Engineering (Information Technology)

Prof. J. K. Kamble
Lab Teacher
Department of Information Technology

Dr. A. S. Ghotkar
Head of Department
Department of Information Technology

Place:
Date:

ACKNOWLEDGEMET

We thank everyone who has helped and provided valuable suggestions for successfully creating a wonderful project.

We are very grateful to our guide, Prof. J. K. Kamble, Head of Department Dr. A. S. Ghotkar and our principal Dr. S.T. Gandhe. They have been very supportive and have ensured that all facilities remained available for smooth progress of the project.

We would like to thank our professor and Prof. J. K. Kamble for providing very valuable and timely suggestions and help.

Rohit pendse
Harshal Rajput
Nisha Rudrawar
Bhavana Sanap
Satyajeet Sankpal

III

ABSTRAT

This “Multiplayer Chess Game” project presents the development of a distributed systems multiplayer chess game using the Swing user interface toolkit and Remote Method Invocation (RMI). The report covers the architecture, design, and implementation of the game, including the use of the Model-View-Controller (MVC) design pattern. The server and client components are implemented using Swing, and the RMI interface is used to communicate between the two components. The report also discusses the challenges faced during the development process and the solutions implemented to overcome them. The end product is a fully functional, interactive, and engaging chess game that can be played by multiple players over a network. This report provides a comprehensive guide for building a similar distributed systems multiplayer game using Swing and RMI.

LIST OF FIGURES

LIST OF TOPIC

INTRODUCTION

Distributed systems have transformed the way software applications are designed, developed, and deployed. One such application is multiplayer games, which enable players to compete against each other over a network. In this project , we present the development of a multiplayer chess game using the Swing user interface toolkit and Remote Method Invocation (RMI). The game is designed to allow multiple players to play against each other over a network, with the server managing the game state and the clients providing the user interface for players to interact with the game. It covers the architecture, design, and implementation of the game, including the use of the Model-View-Controller (MVC) design pattern. We also discuss the challenges faced during the development process and the solutions implemented to overcome them. The end product is a fully functional, interactive, and engaging chess game that can be played by multiple players over a network.

Chess is a two-person game in which each player controls 16 pieces across a checkerboard in an effort to checkmate the king of the other player. Chess is implemented in our project with a graphical user interface. The chess game adheres to the fundamental chess principles, and each chess piece can only move in accordance with movements that are legitimate for that piece.

When playing chess, the pieces are what you move on the chessboard. Chess pieces come in six different varieties. Eight pawns, two bishops, two knights, two rooks, one queen, and one king make up each side's initial set of 16 pieces.

The Pawn

When a game begins, each side starts with eight pawns. White's pawns are located in the second rank, while Black's pawns are located in the seventh rank. The pawn is the least powerful piece and is worth one point. If it is a pawn's first move, it can move forward one or two squares. If a pawn has already moved, then it can move forward just one square at a time. It attacks (or captures) each square diagonally to the left or right. In the following diagram, the pawn has just moved from the e2-square to the e4-square and attacks the squares d5 and f5.

The Bishop

Each side starts with two bishops, one on a light square and one on a dark square. When a game begins, White's bishops are located on c1 and f1, while Black's bishops are located on c8 and f8. The bishop is considered a minor piece (like a knight) and is worth three points. A bishop can move diagonally as many squares as it likes, as long as it is not blocked by its own pieces or an occupied square. An easy way to remember how a bishop can move is that it moves like an "X" shape. It can capture an enemy piece by moving to the occupied square where the piece is located.

The Knight

Each side starts with two knights—a king's knight and a queen's knight. When a game starts, White's knights are located on b1 and g1, while Black's knights are located on b8 and g8. The knight is considered a minor piece (like a bishop) and is worth three points. The knight is the only piece in chess that can jump over another piece! It moves one square left or right horizontally and then two squares up or down vertically,

OR it moves two squares left or right horizontally and then one square up or down vertically—in other words, the knight moves in an "L-shape." The knight can capture only what it lands on, not what it jumps over.

The Rook

Each side starts with two rooks, one on the queenside and one on the kingside. All four rooks are located in the corners of the board. White's rooks start the game on a1 and h1, while Black's rooks are located on a8 and h8. The rook is considered a major piece (like the queen) and is worth five points. It can move as many squares as it likes left or right horizontally, or as many squares as it likes up or down vertically (as long as it isn't blocked by other pieces).

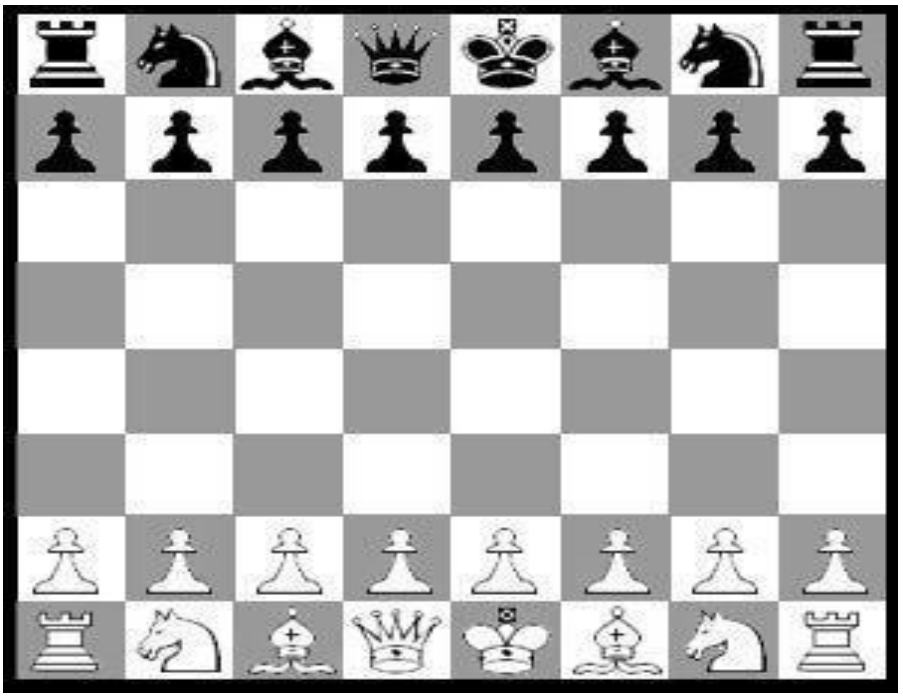
The Queen

The queen is the most powerful chess piece! When a game begins, each side starts with one queen. The white queen is located on d1, while the black queen is located on d8. The queen is considered a major piece (like a rook) and is worth nine points. It can move as many squares as it likes left or right horizontally, or as many squares as it likes up or down vertically (like a rook). The queen can also move as many squares as it likes diagonally (like a bishop).

The King

The king is the most important chess piece. The goal of a game of chess is to checkmate the king. When a game starts, each side has one king. White's king is located on e1, while Black's king starts on e8. The king is not a very powerful piece, as it can only move (or capture) one square in any direction. When a king is attacked, it is called a "check.

Fig 1 : Chess Board



SCOPE AND OBJECTIVE

Scope :

This multiplayer Chess game uses DS concepts and developed using java programming. There is a game architecture, threaded server, java swing for gui elements. Both Players get smooth experience while playing this game.

Objective :

- Both players will able to see when is their turn for playing.
- Only blocks available to play according to rules are allowed to place chess pieces.
- The chess pieces killed will be visible to players.
- Smooth playing experience for players without miscommunication.

SYSTEM ARCHITECTURE/PROJECT FLOW:

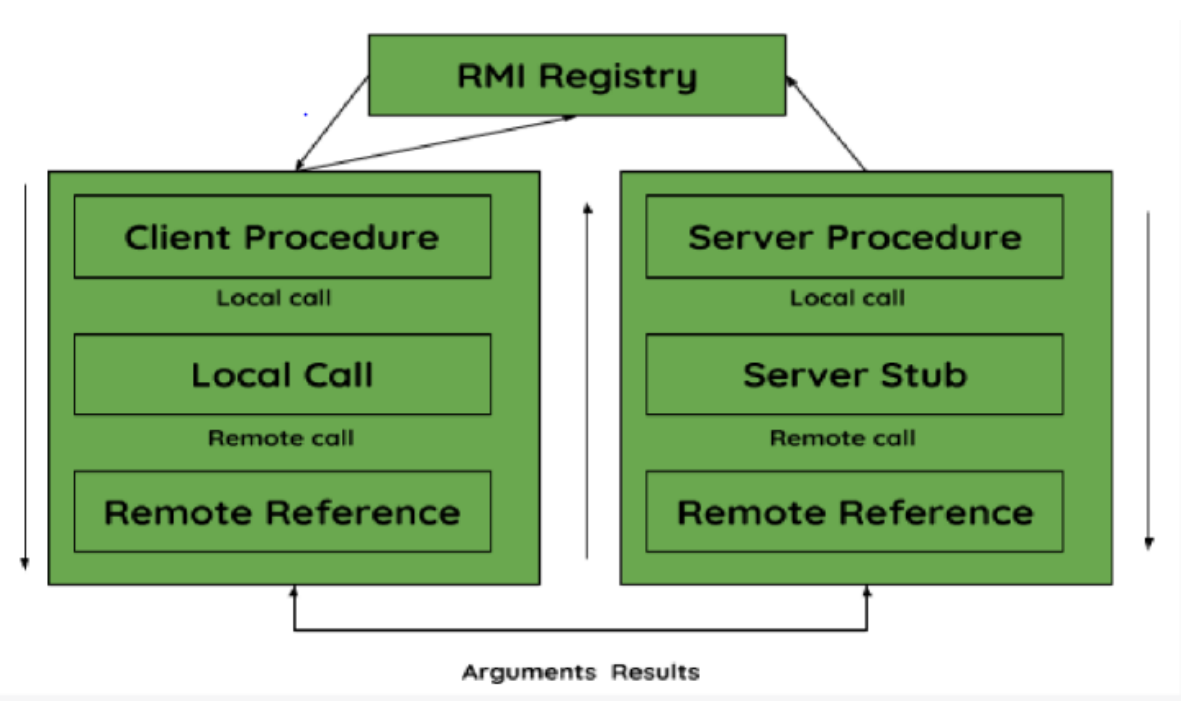


Fig. 2 System Architecture

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

CODE AND SNAPSHOT

Client

Addressframe.java

```
import javax.swing.*;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;


public class AddressFrame extends JFrame {
   //declare components globally so they can be edited from any event handler
   private static final long serialVersionUID = 1L; //once again this is needed
   JPanel pan; //panel for components
   JLabel lab; //label
   JTextField tf; //text field
   JButton button; //enter button

   public AddressFrame() {
      //set size of window
      this.setSize(400, 65);

      //use awt.Toolkit to get screensize and set xPos and yPos to the center of
the screen
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension dim = tk.getScreenSize();
      int xPos = (dim.width / 2) - (this.getWidth() / 2);
      int yPos = (dim.height / 2) - (this.getHeight() / 2);
      //set location to center defined by xPos and yPos
      this.setLocation(xPos, yPos);

      //turn off resizability, set close operation, turn off decoration, and set title
of window
      this.setResizable(false);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.setTitle("Enter IP Address of Server");

      //create panel
      pan = new JPanel();
```

```
        //Create components

        //label
        lab = new JLabel("Enter IP Address of Server");
        lab.setToolTipText("In the box below, please enter the IP address " +
                "of the Server hosting the game. This should be in" +
                "format: \"XXX.XXX.XXX.XXX\" where every x is a number");

        //text field
        tf = new JTextField("", 10);
        tf.setToolTipText("Enter the IP address here with the format: " +
                "\"XXX.XXX.XXX.XXX\" where every x is a number");
        //button
        button = new JButton("Enter");
        button.setToolTipText("Press to confirm IP address entered and connect to
Server");
        ButtonListener butListener = new ButtonListener();
        button.addActionListener(butListener);



        //add components to panel and then add the panel to the Frame
        pan.add(lab);
        pan.add(tf);
        pan.add(button);
        this.add(pan);

        this.setVisible(false);

    }

    //define button listener class
    private class ButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (e.getSource() == button) {
                ChessClient.serverIP = tf.getText();
                ChessClient.ipEntered = true;
            }
        }
    }
```

```
}
```

ButtonFrame.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.rmi.RemoteException;

public class ButtonListener implements ActionListener {
   @Override
   public void actionPerformed(ActionEvent e) {
      //for SessionFrame
      if (e.getSource() == SessionFrame.game1JoinButton) {
         ChessClient.gameSessionSelected = 0;
      }
      if (e.getSource() == SessionFrame.game2JoinButton) {
         ChessClient.gameSessionSelected = 1;
      }
      if (e.getSource() == SessionFrame.game3JoinButton) {
         ChessClient.gameSessionSelected = 2;
      }


      //for ChessClient
      if (e.getSource() == ChessClient.quitBut) {
         try {
            ChessClient.gameExited = true;
            ChessClient.exitGame();
         } catch (RemoteException e1) {
            e1.printStackTrace();
         } catch (InterruptedException e1) {
            e1.printStackTrace();
         }
      }

      for (int i = 0; i <= 7; i++) {
         for (int j = 0; j <= 7; j++) {
            if (e.getSource() == ChessClient.buttonArray[i][j]) {
               ChessClient.selectionX = i;
               ChessClient.selectionY = j;
               ChessClient.pieceSelected = true;
```

```
            }
        }

    }
  }
}
```

Server

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class GameSession extends UnicastRemoteObject implements
GameSessionInterface
{
        //not sure what this does, but it's required to not give warnings
        private static final long serialVersionUID = 1L;
        private int playersInGame = 0;
        private Board board = new Board();

        public GameSession() throws RemoteException
        {

        }

        @Override
        public int getOccupancy() throws RemoteException
        {
                return playersInGame;
        }

        @Override
        public String getPieceTypeAt(int x, int y) throws RemoteException
        {
                return board.getTypeAt(x, y);
        }

        @Override
```

```
public boolean playerJoin() throws RemoteException
{
	if(playersInGame == 0)
	{
		playersInGame++;
		return true;
	}
	else if(playersInGame == 1)
	{
		playersInGame++;
		board.setWinningPlayer(0);
		return true;
	}
	else{return false;}
}

@Override
public int getColorAt(int x, int y) throws RemoteException
{
	return board.getColorAt(x, y);
}

@Override
public boolean gameOver() throws RemoteException
{
	return board.gameOver();
}

@Override
public int getWinningPlayer() throws RemoteException
{
	return board.getWinningPlayer();
}

@Override
public int getPlayerTurn() throws RemoteException
{
	return board.getPlayerTurn();
}

@Override
public void nextPlayerTurn() throws RemoteException
```

```
	{
		board.nextPlayerTurn();
	}

	@Override
	public void playerExit() throws RemoteException
	{
		if(playersInGame == 1)
		{
			playersInGame--;

		}
		else if(playersInGame == 2)
		{
			playersInGame--;
		}
		else
		{
			playersInGame = 0;
		}

	}

	@Override
	public void resetBoard() throws RemoteException
	{
		board.populateBoard();
	}

	@Override
	public void setWinningPlayer(int p) throws RemoteException
	{
		board.setWinningPlayer(p);

	}

	@Override
	public boolean swapPiece(int x1, int y1, int x2, int y2, int c) throws
RemoteException
	{
		return board.swap(x1, y1, x2, y2, c);
	}
```

```
	@Override
	public void setPlayerTurn(int t) throws RemoteException
	{
		board.setPlayerTurn(t);
	}
}
```

RESULT

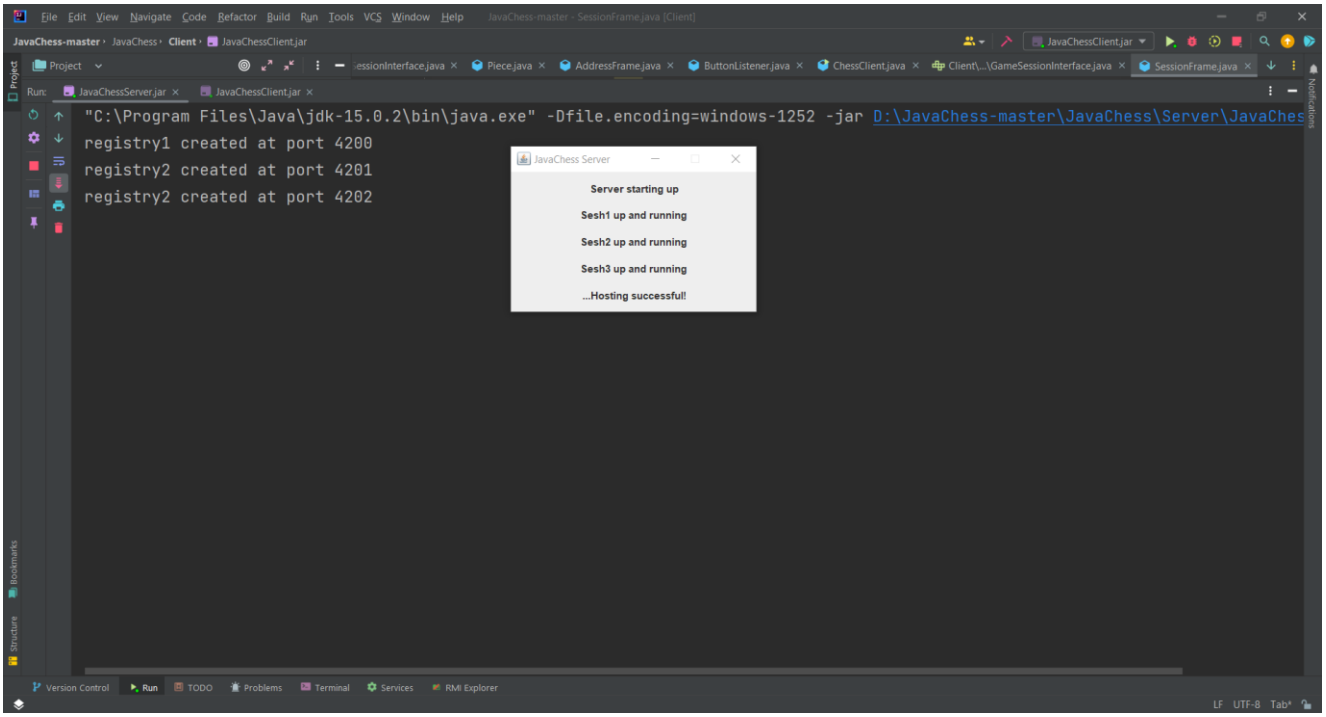


fig.3 Server

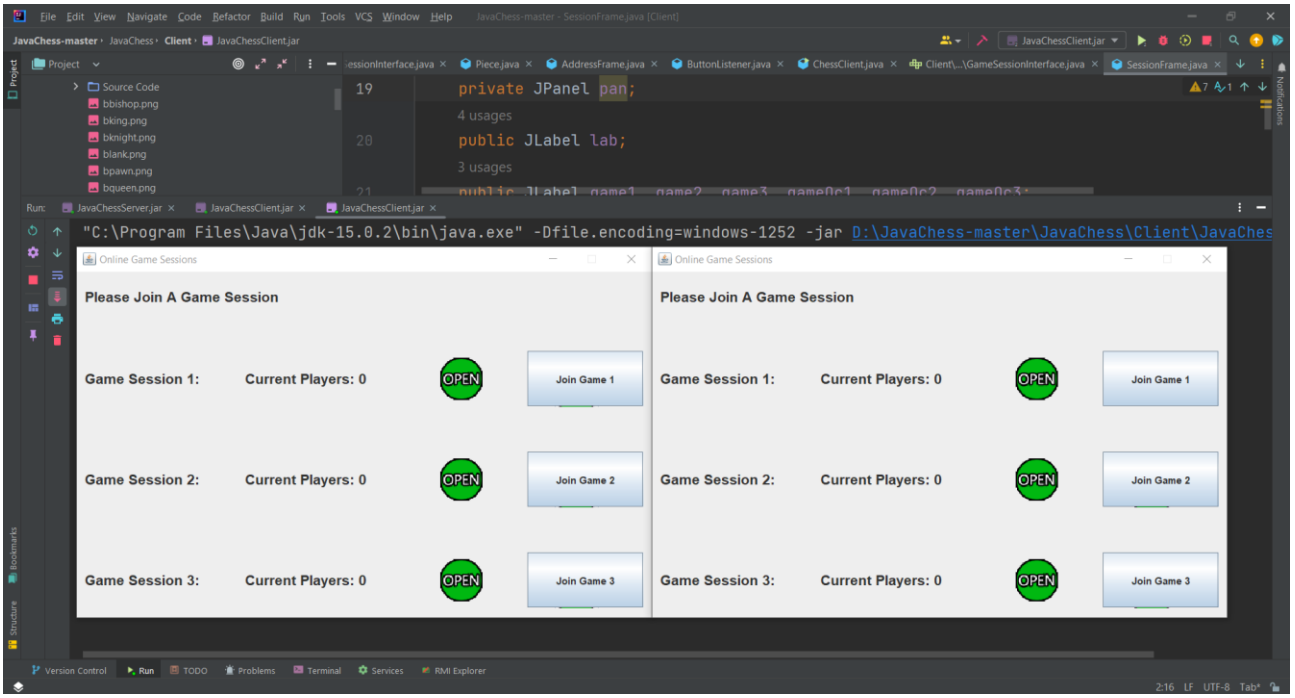


fig. Both clients started

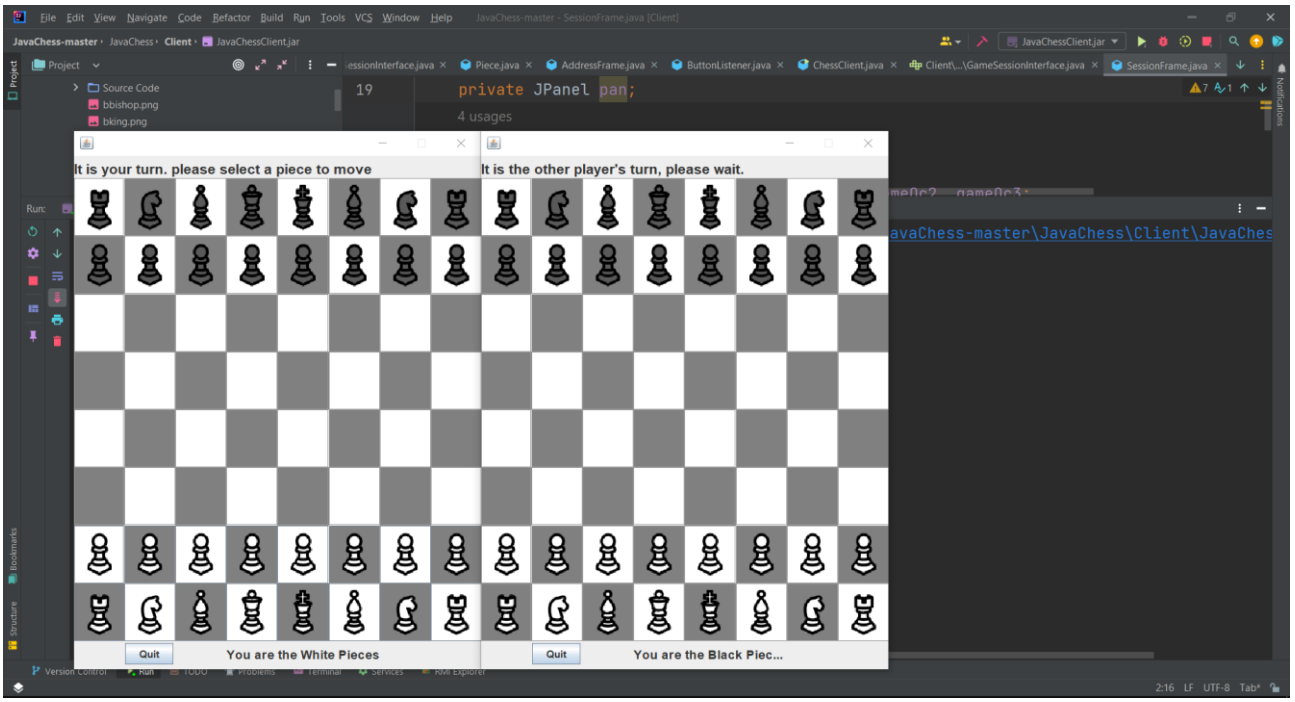


fig.4 Both clients joined

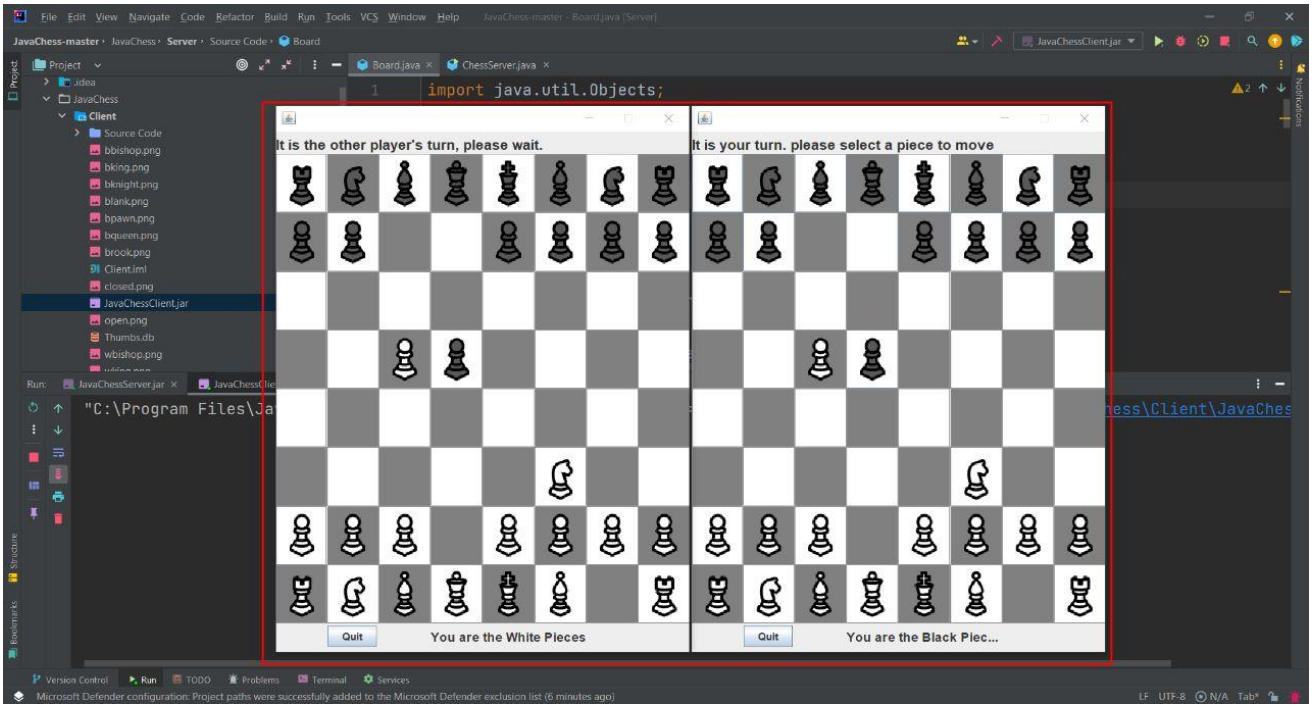


fig.5 final output

CONCLUSION AND FUTURE SCOPE

The threaded architecture allows for multiple players to interact with the game simultaneously. Each player will have their own thread, enabling them to make moves independently and providing a more interactive multiplayer experience.

With the use of threads, players can make moves concurrently, allowing for a more dynamic and responsive gameplay experience. Players can take their turns without waiting for other players to complete their moves.

The threaded architecture provides a foundation for scalability. It allows for future enhancements such as adding more players, incorporating network multiplayer capabilities, or integrating AI opponents without significantly modifying the existing codebase.

Future Scope:

1. You can extend the game by adding features such as move hints, undo/redo functionality, time management, different game modes (e.g., timed matches, puzzle challenges), and support for multiple board themes.
2. Implement the ability to save and load game states, allowing players to resume games at a later time. This feature can include options to save games locally or in a database.
3. Implement a system to track player statistics, achievements, and high scores. This can add a competitive element to the game and motivate players to improve their skills.

REFERENCES

[1] http://chessboardjs.com/

[2] https://github.com/jhlywa/chess.js

[3]https://docs.mongodb.com/manual/tutorial/conf