CMPT 371
SPRING 2025

FINAL **PROJECT**

GROUP 9
KRISH BEDI (301563666)
GOURAV SHARMA (301475592)
RUBEN DUA (301540990)
PRATHAM GARG (301565031)

Table of Contents

Table of Contents **2**
1. Introduction **3**
2. Game Overview **3**
3. Architecture **3**
Key Components: 3
4. Concurrency Control **4**
5. Message Types **4**
6. Code Highlights **5**
7. Limitations and Future Improvements **5**
8. Screenshots: **5**
- Welcome Screen 5
- Game Rules 6
- The 8x8 game grid before and during gameplay 6
- Color picker dialog popup 7
- Sequence of two clients interacting with the same cell 8
- Winning condition 8

9. Code Snippets: **9**
- Server 9
- Client 9
- Shared Object Handling 10

10. Conclusion **12**
11. Contribution - Group 9 .... **12**
12. Links **12**

1. Introduction

Deny and Conquer is a Java-based multiplayer game that combines real-time interaction with concurrency control in a shared game board environment. Players compete to claim cells in an 8x8 grid by drawing in them, with the goal of capturing more territory than others. The game uses Java Swing for the user interface and Java Sockets for client-server communication.

2. Game Overview

The game, Deny and Conquer, is a multiplayer collaborative/drawing game where players compete to claim cells on a game board. The game unfolds as follows:

- Each player selects a username and their unique color at the start of the game.
- The game board consists of an 8x8 grid of cells, where each cell is 50x50 pixels.
- Players can click and drag on a cell to "claim" it. To claim the cell, they must "draw" over at least 50% of the cell's area.
- A claimed cell is marked by the claiming player's color, and it cannot be modified by others.
- Once all cells are claimed, the game ends, and the player with the most claimed cells is declared the winner. If there is a tie in the number of claimed cells, the game ends in a draw.

3. Architecture

The game uses a client-server architecture to manage multiplayer interactions and maintain a consistent game state. The server is responsible for handling all client connections, managing the game board, and broadcasting messages to keep every client in sync. Each client represents a player, providing a graphical interface to draw on the board, send actions to the server, and receive updates from other players.

Key Components:

- **GameBoard** and **Cell**: Together, these classes represent the game grid. They manage the state of each cell, including drawing pixels, checking if a cell is claimable, and handling reset logic.
- **GamePanel**: Listens for player input through mouse events and triggers the appropriate lock, draw, or release actions.

- **Client** and **Server**: Handle socket communication using serialized Java objects. The server processes player messages and rebroadcasts them to all clients.
- **MessageToSend**: A serializable object used to package all game-related data (like cell coordinates, pixel position, player color, and action type) for transmission between client and server.

4. Concurrency Control

In Deny and Conquer, concurrency control is achieved through a lock-based coordination mechanism to ensure that only one player can interact with a given cell at any time. When a player clicks on a cell to begin drawing, their client sends a "RequestLock" message to the server, specifying the cell's row, column, and pixel coordinates. The server checks whether the cell is available using a ReentrantLock, and either grants the lock (responding with "LockGranted") or denies it ("LockNotGranted"). If the lock is granted, the client can proceed with drawing on the cell and sends "Scribble" messages to update the board in real time across all clients. Once the drawing is complete, the client checks if the cell has been filled with at least 125 pixels. If successful, a "Filled" message is sent and the cell becomes permanently claimed. If not, a "NotFilled" message is sent and the server calls unlock() on the cell, making it available again. This approach ensures proper access control using actual mutual exclusion (mutex) while maintaining smooth and synchronized gameplay across all clients in a distributed network.

5. Message Types

- **RequestLock**: Sent by a client to the server to request access to draw on a specific cell.
- **LockGranted**: Sent by the server to confirm that the client may proceed with drawing.
- **LockNotGranted**: Sent by the server when the requested cell is already in use by another player.
- **Scribble**: Represents a pixel drawn by a player during the drawing phase.
- **Filled**: Sent when a player successfully fills enough of a cell (125 pixels), making the claim permanent.
- **NotFilled**: Sent when a player does not fill enough of the cell; the lock is released and drawing is cleared.
- **GameOver**: Sent by the server when all cells are claimed, declaring the winner.
- **Draw**: Indicates a tie in the number of claimed cells between players.

6. Code Highlights

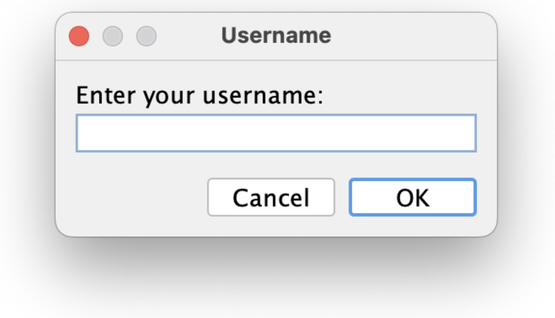
- **Cell.java**: Manages the behavior of individual cells, including pixel drawing, checking claim thresholds, and tracking lock ownership via ReentrantLock. It exposes tryLock() and unlock() methods to ensure exclusive access per client.
- **GamePanel.java**: Handles mouse input and user interaction. Sends RequestLock messages when a player clicks a cell, processes LockGranted responses, and coordinates the draw/release cycle.
- **Client.java**: Establishes connection to the server, sends structured game messages, and listens for broadcasts. Processes updates like Scribble, LockGranted, and GameOver to update the UI and local game state.
- **Server.java**: Controls the overall game state and ensures mutual exclusion on shared resources. Handles incoming requests with locking logic, and broadcasts validated messages (e.g., Scribble, Filled) to all connected clients.

7. Limitations and Future Improvements

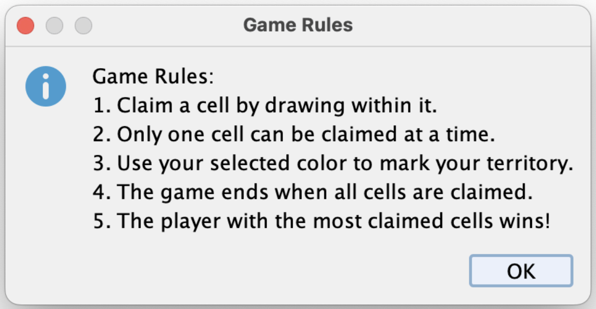
- Add a scoreboard to track claimed cells per player.
- Add a player join/leave notification system.
- Enhance visual effects and sound.
- Add authentication.

8. Screenshots:

- Welcome Screen



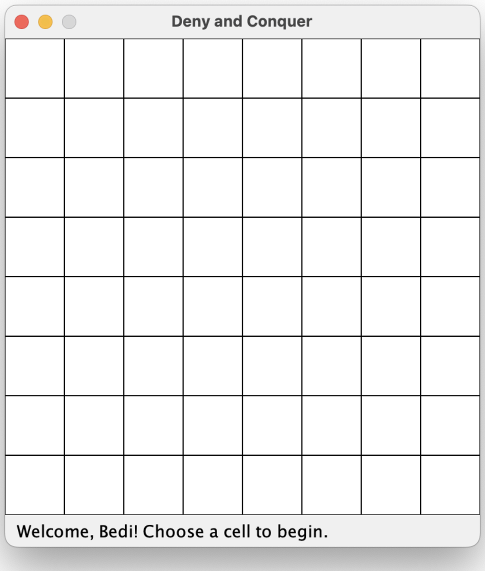
- Game Rules



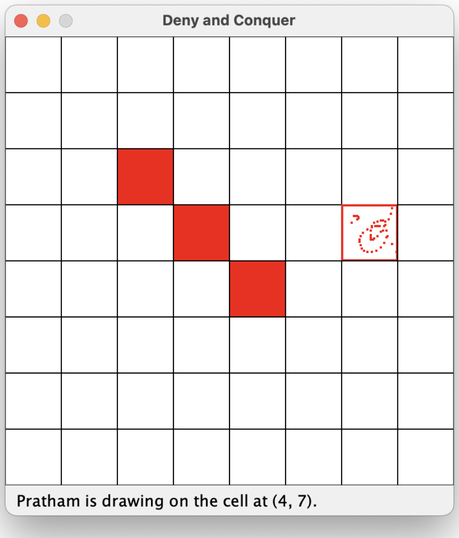
Game Rules

Game Rules:
1. Claim a cell by drawing within it.
2. Only one cell can be claimed at a time.
3. Use your selected color to mark your territory.
4. The game ends when all cells are claimed.
5. The player with the most claimed cells wins!

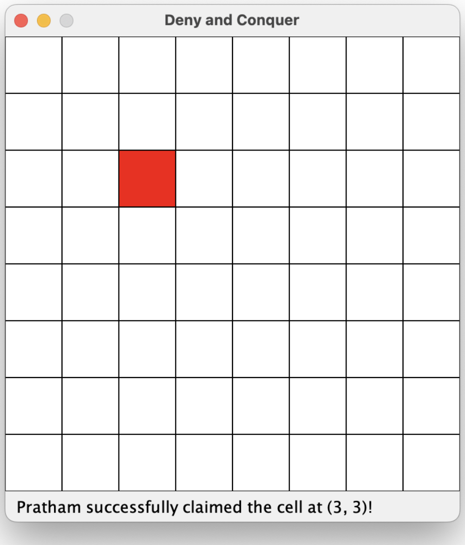
- The 8x8 game grid before and during gameplay



Welcome, Bedi! Choose a cell to begin.

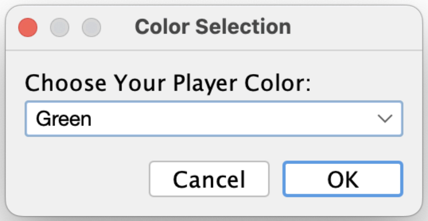


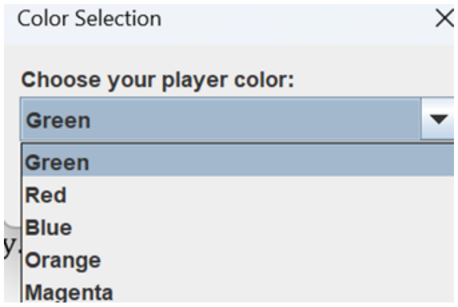
Pratham is drawing on the cell at (4, 7).



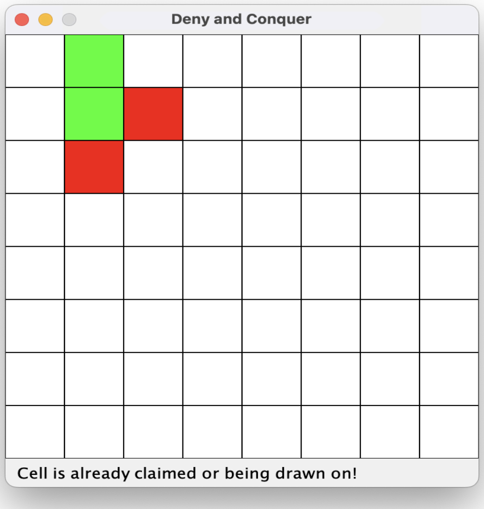
Pratham successfully claimed the cell at (3, 3)!

- Color picker dialog popup



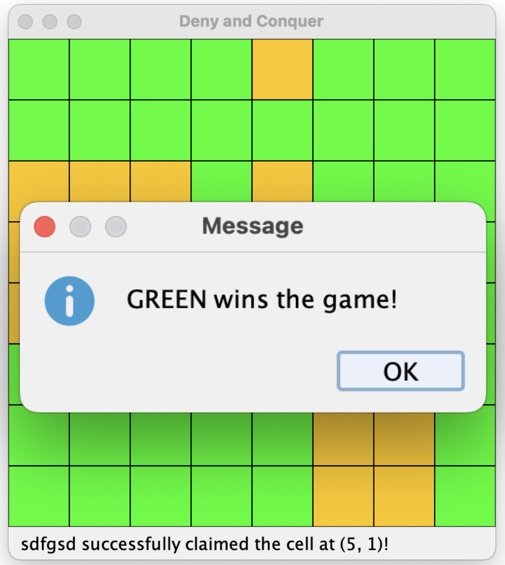


- Sequence of two clients interacting with the same cell



Cell is already claimed or being drawn on!

- Winning condition

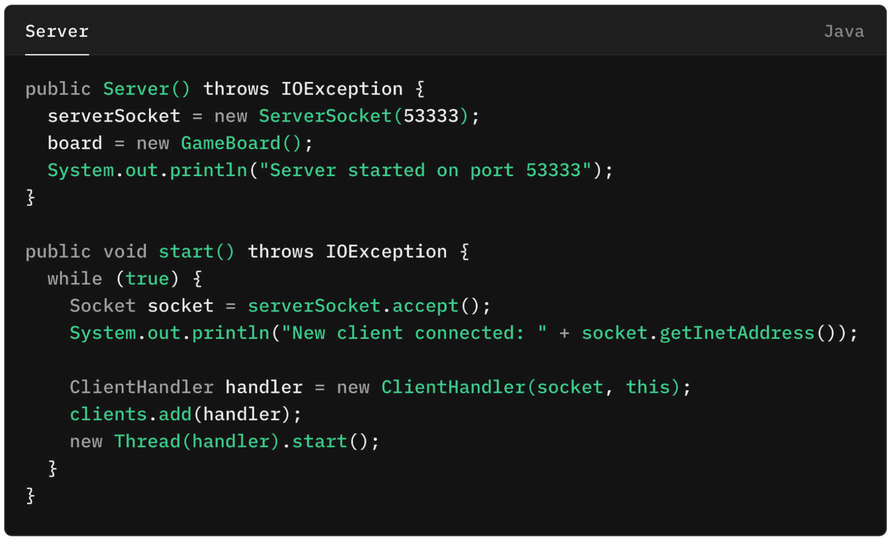


GREEN wins the game!

sdfgsd successfully claimed the cell at (5, 1)!

9. Code Snippets:

- Server



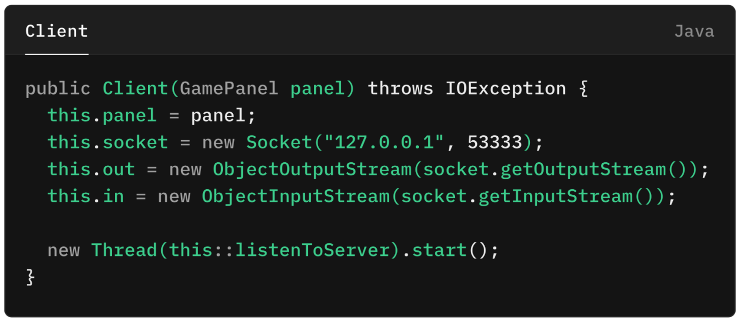
Server Java

```java
public Server() throws IOException {
  serverSocket = new ServerSocket(53333);
  board = new GameBoard();
  System.out.println("Server started on port 53333");
}

public void start() throws IOException {
  while (true) {
    Socket socket = serverSocket.accept();
    System.out.println("New client connected: " + socket.getInetAddress());

    ClientHandler handler = new ClientHandler(socket, this);
    clients.add(handler);
    new Thread(handler).start();
  }
}
```

- Client



Client Java

```java
public Client(GamePanel panel) throws IOException {
  this.panel = panel;
  this.socket = new Socket("127.0.0.1", 53333);
  this.out = new ObjectOutputStream(socket.getOutputStream());
  this.in = new ObjectInputStream(socket.getInputStream());

  new Thread(this::listenToServer).start();
}
```

- ## Shared Object Handling

Shared-Object-Handling Java

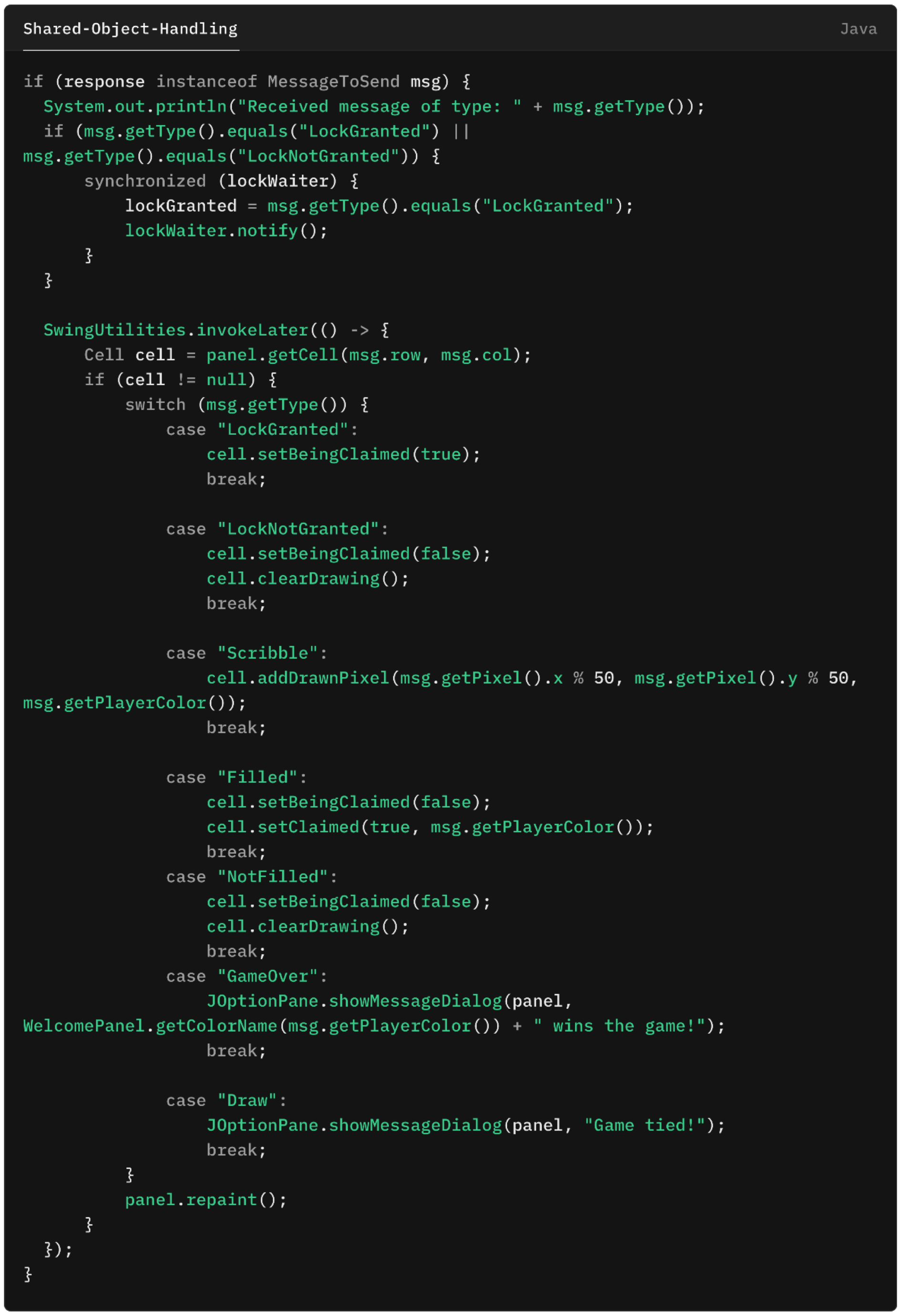
```java
@Override
public void run() {
    try {

        out = new ObjectOutputStream(socket.getOutputStream());
        out.flush(); // Ensure the stream header is sent
        in = new ObjectInputStream(socket.getInputStream());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    try {
        Object obj;
        while ((obj = in.readObject()) != null) {
            if (obj instanceof MessageToSend message) {

                if (clientId == null) {
                    clientId = message.getSenderID();
                }

                if (message.getType().equals("RequestLock")) {
                    System.out.println("Received message: " + message.getType());
                    server.grantLock(message);
                } else if (message.getType().equals("NotFilled")) {
                    System.out.println("Received message: " + message.getType());
                    server.unlockCell(message.getRow(), message.getCol());
                    server.broadcast(message);
                }
                else {
                    server.broadcast(message);
                }
            }
        }
    } catch (EOFException e) {
        // Client closed connection
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            // Ignore
        }
        server.remove(this);
        System.out.println("Client disconnected: " + socket.getInetAddress());
    }
}
```



Shared-Object-Handling

Java

```java
if (response instanceof MessageToSend msg) {
  System.out.println("Received message of type: " + msg.getType());
  if (msg.getType().equals("LockGranted") ||
msg.getType().equals("LockNotGranted")) {
      synchronized (lockWaiter) {
          lockGranted = msg.getType().equals("LockGranted");
          lockWaiter.notify();
      }
  }

  SwingUtilities.invokeLater(() -> {
      Cell cell = panel.getCell(msg.row, msg.col);
      if (cell != null) {
          switch (msg.getType()) {
              case "LockGranted":
                  cell.setBeingClaimed(true);
                  break;

              case "LockNotGranted":
                  cell.setBeingClaimed(false);
                  cell.clearDrawing();
                  break;

              case "Scribble":
                  cell.addDrawnPixel(msg.getPixel().x % 50, msg.getPixel().y % 50,
msg.getPlayerColor());
                  break;

              case "Filled":
                  cell.setBeingClaimed(false);
                  cell.setClaimed(true, msg.getPlayerColor());
                  break;
              case "NotFilled":
                  cell.setBeingClaimed(false);
                  cell.clearDrawing();
                  break;
              case "GameOver":
                  JOptionPane.showMessageDialog(panel,
WelcomePanel.getColorName(msg.getPlayerColor()) + " wins the game!");
                  break;

              case "Draw":
                  JOptionPane.showMessageDialog(panel, "Game tied!");
                  break;
          }
          panel.repaint();
      }
  });
}
```

The shared object handling code consists of two main parts: client-side UI updates and server-side message processing. On the client side, incoming messages from the server are processed using SwingUtilities.invokeLater() to safely update the user interface. Depending on the message type (e.g., "Lock", "Scribble", "Filled", "Unlock", "GameOver"), the client updates the corresponding cell's state, such as marking it as being claimed, adding pixel data, finalizing ownership, or resetting it, and then repaints the board. On the server side, each connected client is managed by a ClientHandler thread, which listens for serialized MessageToSend objects. When a player requests a lock, the server checks if the cell can be locked and responds accordingly with "LockGranted" or "LockNotGranted". If a player fails to meet the threshold for a valid claim, a "NotFilled" message is processed, and the server unlocks the cell. Other messages, such as drawing actions or game results, are broadcast to all clients to keep their game boards in sync.

10. Conclusion

This project helped in understanding practical implementation of client-server systems, Java concurrency, and building user-friendly graphical applications. It demonstrates real-time coordination between multiple players and ensures fairness using locking mechanisms. Future improvements could include player scoring, chat integration, and mobile compatibility.

11. Contribution - Group 9

Gourav Sharma (301475592) - 25%
Ruben Dua (301540990) - 25%
Krish Bedi (301563666) - 25%
Pratham Garg (301565031) - 25%

12. Links

Github: https://github.com/krishbedi17/Deny-Conquer

Youtube: https://youtu.be/Ty4GEiwkbSc