INFO0010: Project 1

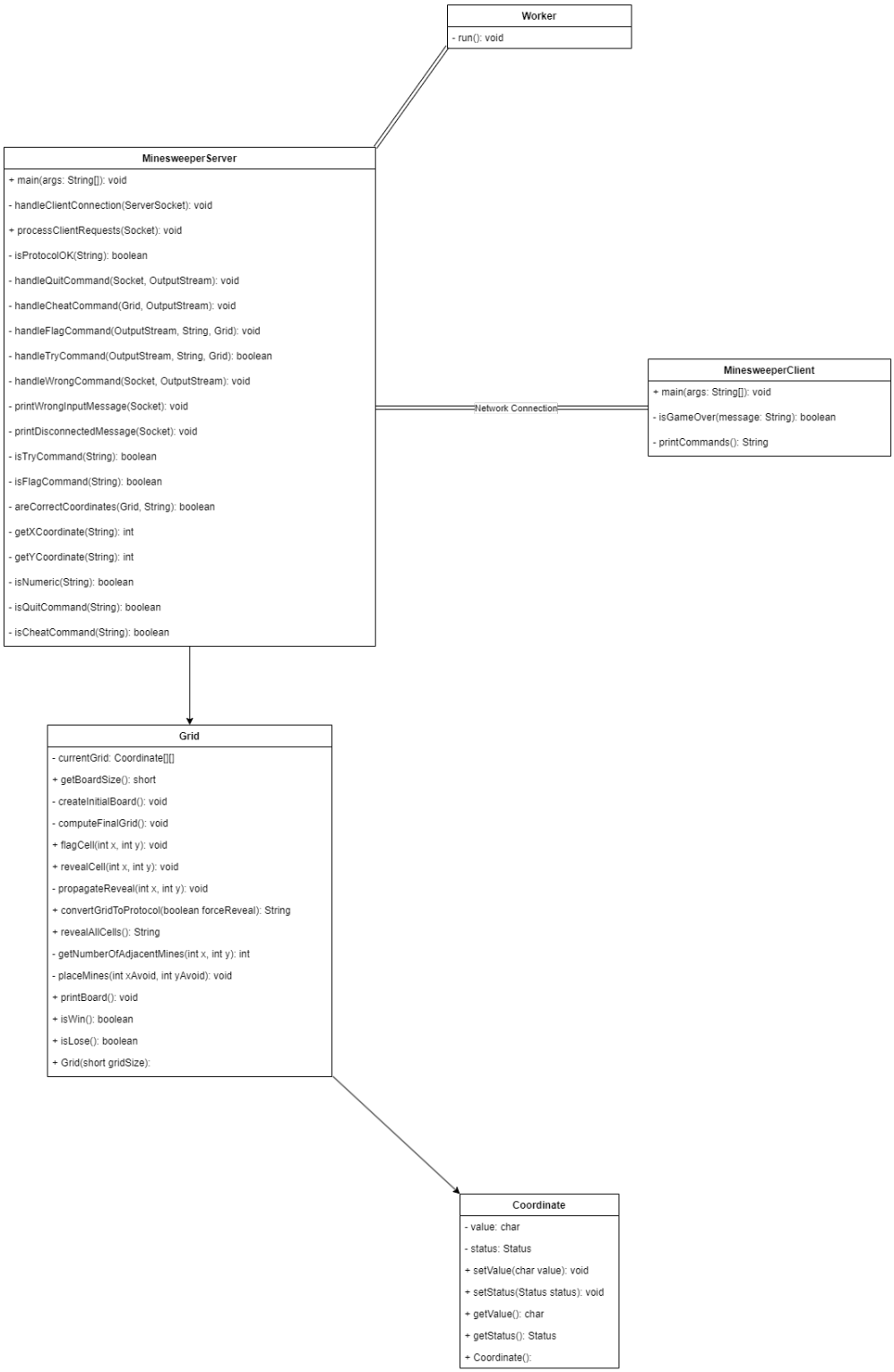
EL MASRI Sam

21 October 2024

Contents

1 Software Architecture

The software architecture is described by this class diagram :



2 Program Logic

2.1 How the Minesweeper Server Works:

The `MinesweeperServer` operates by managing client connections and processing Minesweeper-related commands over a network. Below is a sequential breakdown of how the server operates, along with the relevant code locations in the source.

2.2 Server Initialization:

- The server starts by creating a `ServerSocket` on a specific port (2377).
- It enters an infinite loop, waiting for incoming client connections.
- This is implemented in the `main` method:

```
public static void main(String[] args) throws IOException {
    try(ServerSocket serverSocket = new ServerSocket(SERVER_PORT)) {
        System.out.println("New server socket started on port " + SERVER_PORT);
        while (true) {
            handleClientConnection(serverSocket);
        }
    }
}
```

2.3 Handling Client Connections:

- The server listens for client connections using `serverSocket.accept()`. Once a connection is accepted, a new thread (in the `Worker` class) is created to handle the client's requests.
- This is done in the `handleClientConnection` method:

```
private static void handleClientConnection(ServerSocket serverSocket) {
    try {
        Socket clientSocket = serverSocket.accept();
        if (clientSocket.isConnected()) {
            System.out.println("Client " + clientSocket.getPort() +
            " connected.");
        }
        Worker worker = new Worker(clientSocket);
        worker.start();
    }
}
```

2.4 Processing Client Requests:

- In each worker thread, the server waits for commands from the client, reads the data, and processes it based on the client's input. Commands include `TRY`, `FLAG`, `CHEAT`, and `QUIT`. The data exchange occurs through input/output streams (`InputStream` and `OutputStream`). A new grid is created in each worker thread.
- This is implemented in the `processClientRequests` method:

```
public static void processClientRequests(Socket clientSocket)
    throws IOException
{
    Grid grid = new Grid(GRID_SIZE);
    OutputStream outputServer = clientSocket.getOutputStream();
    InputStream inputClient = clientSocket.getInputStream();
    // Message handling and parsing loop here
}
```

2.5 Message Processing Loop:

- The server reads messages from the client in chunks. It ensures the message ends with the correct protocol (`\r\n\r\n`) before processing.
- Once a valid message is received, the server checks if it matches any of the predefined commands: `QUIT`, `CHEAT`, `FLAG`, or `TRY`.
- This logic is in the same `processClientRequests` method.

2.6 Handling Specific Commands:

- **QUIT Command:** Ends the connection and sends a goodbye message. Implemented in `handleQuitCommand`.
- **CHEAT Command:** Reveals all mines on the grid. Implemented in `handleCheatCommand`.
- **FLAG Command:** Flags a cell at specified coordinates. Implemented in `handleFlagCommand`.
- **TRY Command:** Attempts to reveal a cell and checks if the game is over. Implemented in `handleTryCommand`.

2.7 Timeout Management:

- The server sets a timeout for each client connection. If the client is inactive for more than 60 seconds, the connection is automatically closed.
- This is done by calling:

```
clientSocket.setSoTimeout(INACTIVE_TIME_OUT); // 60-second timeout
```

2.8 Command Validation:

- For commands like `TRY` and `FLAG`, the server validates whether the coordinates are within the grid. If invalid, it sends an error message (`INVALID RANGE`).
- Validation logic is implemented in helper methods such as `areCorrectCoordinates` and `isProtocolOK`.

The server's operations are organized into methods like `handleQuitCommand`, `handleCheatCommand`, `handleFlagCommand`, and `handleTryCommand`, each implementing specific functionality based on the client's input.

3 TCP Stream

In the `MinesweeperServer` class, requests from the TCP stream are read using an `InputStream` associated with the client socket. Below is a detailed explanation of how the server reads requests from the TCP stream, along with the relevant source code.

3.1 Overview

1. **Establishing Input Stream:** The server retrieves the input stream from the connected client socket. This is done within the `processClientRequests` method where an `InputStream` object is created for reading data sent by the client.

2. **Reading Data:** The server reads data from the input stream in a loop, using a buffer to store the incoming data. The data is read until the server detects a complete message or the client closes the connection.

3. **Checking Protocol:** The server checks whether the received message conforms to the expected protocol format. Specifically, it looks for a proper end-of-message sequence.

3.2 Source Code

Here is the part of the `processClientRequests` method that handles reading requests from the TCP stream:

```
public static void processClientRequests(Socket clientSocket)
    throws IOException
{
    Grid grid = new Grid(GRID_SIZE);
    OutputStream outputServer = clientSocket.getOutputStream();
    // Establishing input stream
    InputStream inputClient = clientSocket.getInputStream();
    try {
        // Set the timeout for the client socket
        clientSocket.setSoTimeout(INACTIVE_TIME_OUT);
        byte[] buffer = new byte[MSG_SIZE];  // Buffer for incoming data
        StringBuilder messageBuilder = new StringBuilder();

        // Loop until the client sends a "QUIT" command
        while (true) {
            int len = inputClient.read(buffer);  // Reading data from the stream
            if (len == -1) break;  // End of stream (client disconnected)

            // Appending read data to messageBuilder
            messageBuilder.append(new String(buffer, 0, len));
            String receivedMessage = messageBuilder.toString();

            // Check for a complete message based on protocol
            if (!isProtocolOK(receivedMessage)) continue;

            receivedMessage = receivedMessage.trim();  // Trim whitespace

            // Handle commands based on the received message
            if (isQuitCommand(receivedMessage)) {
                handleQuitCommand(clientSocket, outputServer);
                break;
            } else if (isCheatCommand(receivedMessage)) {
                handleCheatCommand(grid, outputServer);
            } else if (isFlagCommand(receivedMessage)) {
                handleFlagCommand(outputServer, receivedMessage, grid);
            } else if (isTryCommand(receivedMessage)) {
                boolean isOver =
                    handleTryCommand(outputServer, receivedMessage, grid);
                if (isOver) {
                    System.out.println("Game over for client "
                    + clientSocket.getPort() + " => disconnecting.");
                    clientSocket.close();
                    break;
                }
            } else {
                handleWrongCommand(clientSocket, outputServer);
            }

            // Reset the message builder
            messageBuilder.setLength(0);
        }
    } catch (SocketTimeoutException e) {
        System.out.println("Client " + clientSocket.getPort() + " timed out.");
        clientSocket.close();
    }
}
```

3.3 Explanation of the Code

- **InputStream Creation:**

```
    InputStream inputClient = clientSocket.getInputStream();
```

This line establishes the input stream associated with the client socket, allowing the server to read data sent by the client.

- **Data Reading Loop:**

```
    int len = inputClient.read(buffer);
```

The `read` method reads bytes from the input stream into the `buffer`. The length of the bytes read is returned, allowing the server to know how much data was received.

- **Message Handling:**

```
    // Check for a complete message based on protocol
    if (!isProtocolOK(receivedMessage)) continue;
```

After reading, the data is converted into a string and appended to `messageBuilder`. The server checks if the received message is complete using the `isProtocolOK` method.

- **Command Processing:**

```
    // Handle commands based on the received message
    if (isQuitCommand(receivedMessage)) {
        handleQuitCommand(clientSocket, outputServer);
        break;
    } else if (isCheatCommand(receivedMessage)) {
        handleCheatCommand(grid, outputServer);
    } else if (isFlagCommand(receivedMessage)) {
        handleFlagCommand(outputServer, receivedMessage, grid);
    } else if (isTryCommand(receivedMessage)) {
        boolean isOver =
            handleTryCommand(outputServer, receivedMessage, grid);
        if (isOver) {
            System.out.println("Game over for client "
            + clientSocket.getPort() + " => disconnecting.");
            clientSocket.close();
            break;
        }
    } else {
        handleWrongCommand(clientSocket, outputServer);
    }
```

Depending on the command received (e.g., `QUIT`, `CHEAT`, `FLAG`, `TRY`), the corresponding handler method is called to process the command.

4 Multi-thread Organisation

The Minesweeper server achieves multi-threading by creating a new thread for each client connection. This allows the server to handle multiple clients simultaneously, ensuring that each client's requests are processed independently.

4.1 Multi-threading Implementation

1. **Worker Thread Creation:** When a client connects to the server, a new `Worker` thread is instantiated. This thread is responsible for processing all requests from that particular client.
2. **Thread Management:** Each `Worker` instance handles communication with the client, allowing other clients to connect and be served simultaneously.

4.2 Source Code

Here is the relevant portion of the source code that demonstrates how multi-threading is implemented in the `MinesweeperServer` class:

```
private static void handleClientConnection(ServerSocket serverSocket) {
    try {
        // Accept a client connection
        Socket clientSocket = serverSocket.accept();
        if(clientSocket.isConnected()) {
            System.out.println("Client " + clientSocket.getPort() + " connected.");
        } else {
            System.out.println("Client failed to connect.");
            clientSocket.close();
            return;
        }
        // Create a new worker thread for the client
        Worker worker = new Worker(clientSocket);
        worker.start();  // Start the worker thread to handle the client
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

4.3 Worker Class Implementation

The `Worker`[1] class extends `Thread` and overrides the `run` method, where the client's requests are processed:

```
class Worker extends Thread {
    private Socket clientSocket;

    public Worker(Socket socket) {
        this.clientSocket = socket;
    }

    @Override
    public void run() {
        try {
            // Process client requests
            processClientRequests(clientSocket);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

[1] This class comes from the course (Java Socket.pdf)

4.4 Synchronization

In this implementation, synchronization of threads is not explicitly required for processing client requests, as each worker operates on its own instance of the `Grid` object.

5 Robustness

The robustness of the Minesweeper server is achieved through careful exception handling, resource management, and validation of inputs. By addressing potential errors and ensuring proper resource cleanup, the server can handle unexpected situations gracefully.

5.1 Exception Management Strategy

1. **Catching Exceptions**: The server catches various exceptions that may occur during its operation, such as `IOException`, `SocketTimeoutException`, and other unexpected exceptions. This prevents the server from crashing and allows it to handle errors gracefully.

2. **Error Logging**: Errors are logged using `e.printStackTrace()` to provide feedback on the nature of the exception. This can be crucial for debugging and improving server performance.

3. **Resource Cleanup**: Properly closing sockets and streams in the event of an error ensures that resources are freed and do not lead to resource leaks.

5.2 Source Code for Exception Handling

Here are relevant snippets from the Minesweeper server code that illustrate how exceptions are handled:

```
try (ServerSocket serverSocket = new ServerSocket(SERVER_PORT)) {
    System.out.println("New server socket started on port " + SERVER_PORT);
    while (true) {
        handleClientConnection(serverSocket);
    }
} catch (IOException e) {
    System.err.println("Error starting server: " + e.getMessage());
    e.printStackTrace();
}
```

In the above code, the server socket is wrapped in a try-with-resources statement to ensure it is closed automatically. Any IOException during the server's startup will be caught and logged.

5.3 Handling Client Connections

In the `handleClientConnection` method, exceptions are caught to manage errors when clients attempt to connect:

```
private static void handleClientConnection(ServerSocket serverSocket) {
    try {
        Socket clientSocket = serverSocket.accept();
        // Handle client connection...
    } catch (IOException e) {
        System.err.println("Error accepting client connection: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Here, any issues while accepting a client connection are logged, preventing the server from crashing.

5.4 Handling Client Requests

The `processClientRequests` method also includes exception handling to manage timeouts and other I/O errors during client communication:

```
try {
    // Set the timeout for the client socket
    clientSocket.setSoTimeout(INACTIVE_TIME_OUT);
    // Process client requests...
} catch (SocketTimeoutException e) {
    System.out.println("Client " + clientSocket.getPort() + " timed out.");
    clientSocket.close();
} catch (IOException e) {
    System.err.println("I/O error with client " + clientSocket.getPort() +
    ": " + e.getMessage());
    e.printStackTrace();
}
```

In this snippet, a timeout exception is specifically caught to handle cases where a client does not respond in time. Other I/O exceptions are also logged for further diagnosis.

5.5 Handling Client Inputs

The server performs thorough validation of client inputs to ensure that commands and parameters are within expected ranges and formats. This helps prevent incorrect commands from causing exceptions or undefined behavior.

5.5.1 Command Validation

The server checks whether incoming commands from clients are recognized and formatted correctly. For example, commands like `TRY`, `FLAG`, and `CHEAT` are verified.

```
private static boolean isTryCommand(String input) {
    return input.startsWith(TRY_COMMAND);
}

private static boolean isFlagCommand(String input) {
    return input.startsWith(FLAG_COMMAND);
}

private static boolean isQuitCommand(String input) {
    return input.equals(QUIT_COMMAND);
}

private static boolean isCheatCommand(String input) {
    return input.equals(CHEAT_COMMAND);
}
```

In these methods, the server checks if the incoming command starts with the expected command strings. If the command is invalid, the server responds with an appropriate error message.

5.5.2 Coordinate Validation

In addition to command validation, the server ensures that the coordinates provided by the client are valid and within the grid's boundaries. The following method illustrates this validation:

```
private static boolean areCorrectCoordinates(Grid grid, String input) {
    String[] parts = input.split(" ");
    final int X = 1;
    final int Y = 2;

    if (parts.length == 3) {
        if (!isNumeric(parts[X]) || !isNumeric(parts[Y])) {
            return false;
        }
        int x = getXCoordinate(input);
        int y = getYCoordinate(input);
        // Check if the coordinates are within the grid
        if (x < 0 || x >= grid.getBoardSize() || y < 0 || y >= grid.getBoardSize()) {
            return false;
        }
    } else {
        return false;
    }
    return true;
}
```

In this method: - The input is split into parts, and the server checks if the correct number of parts is provided. - It verifies whether the coordinates are numeric and within the acceptable range of the grid.

6 Conclusion

In summary, the Minesweeper server is designed with a robust architecture that effectively handles multiple client connections through multi-threading. Each client is managed by a dedicated worker thread, enabling simultaneous processing of requests without blocking other clients. The server operates on a clear protocol that governs communication, ensuring that commands are accurately interpreted and executed.

A critical aspect of the server's reliability is its rigorous validation of client inputs. Each command received from the client is checked against predefined formats, ensuring that only valid commands are processed. This validation includes comprehensive checks for the correctness of coordinates and the conformity of messages to the established protocol. By implementing methods such as `areCorrectCoordinates`, the server can effectively reject invalid inputs, thereby preventing potential errors or undefined behaviors that could disrupt game functionality.

Moreover, the server employs a thorough exception management strategy. By catching various exceptions, such as `IOException` and `SocketTimeoutException`, the server maintains its operational integrity even in the face of unexpected conditions. This proactive approach to error handling not only prevents crashes but also allows the server to log useful information for troubleshooting. The use of try-with-resources statements ensures that resources are properly managed and freed, minimizing the risk of resource leaks.

Overall, the Minesweeper server exemplifies a well-structured design that prioritizes robustness, reliability, and user experience. By integrating effective input validation, exception handling, and resource management, the server is prepared to handle a variety of client interactions seamlessly.