Technical Report: Shared Canvas Application

Hanjie Liu

1667156

hanjie.liu@student.unimelb.edu.au

Comp90015 Distrubution System

1. Introduction

1.1 Project Overview

A collaborative drawing application based on java Remote Method Invocation (RMI) was designed and implemented. The application allows multiple users to interact in real time on a shared canvas, with features such as text entry, chat functionality, user access control, and session persistence, saving or loading.

1.2 Development Environment

Stack used: Java, Maven, GSON, RMI, Swing

IDE: IntelliJ
JDK: Java 19
Project run:

1. java –jar server.jar <serverIPAddress> <ServerPort>

2. java –jar client.jar <server-address> <server-port> username

(For testing we can just run without args, it will use default params. The server has a fixed name which is ‘manager’)

2. System Architecture Overview

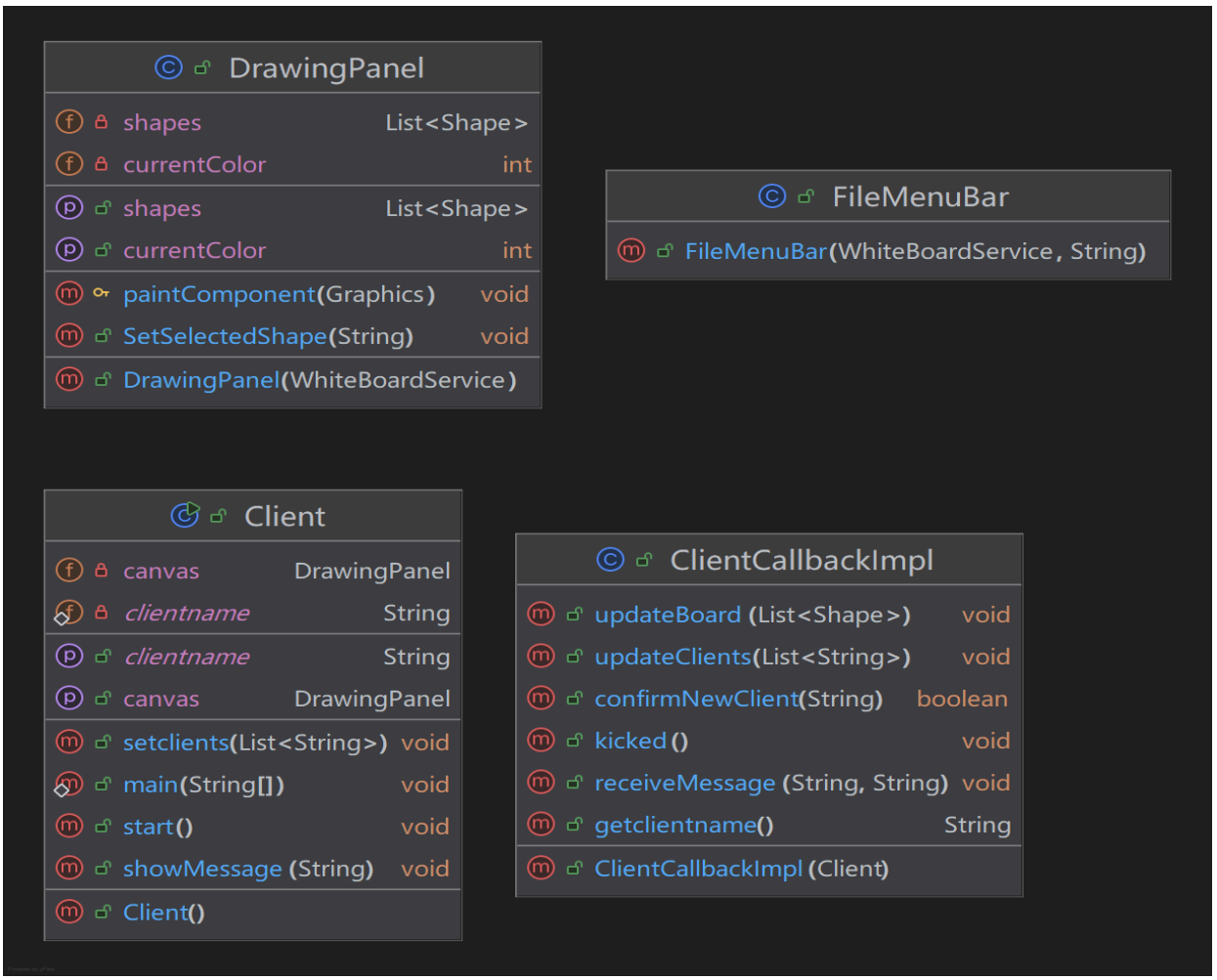
2.1 Codebase Structure

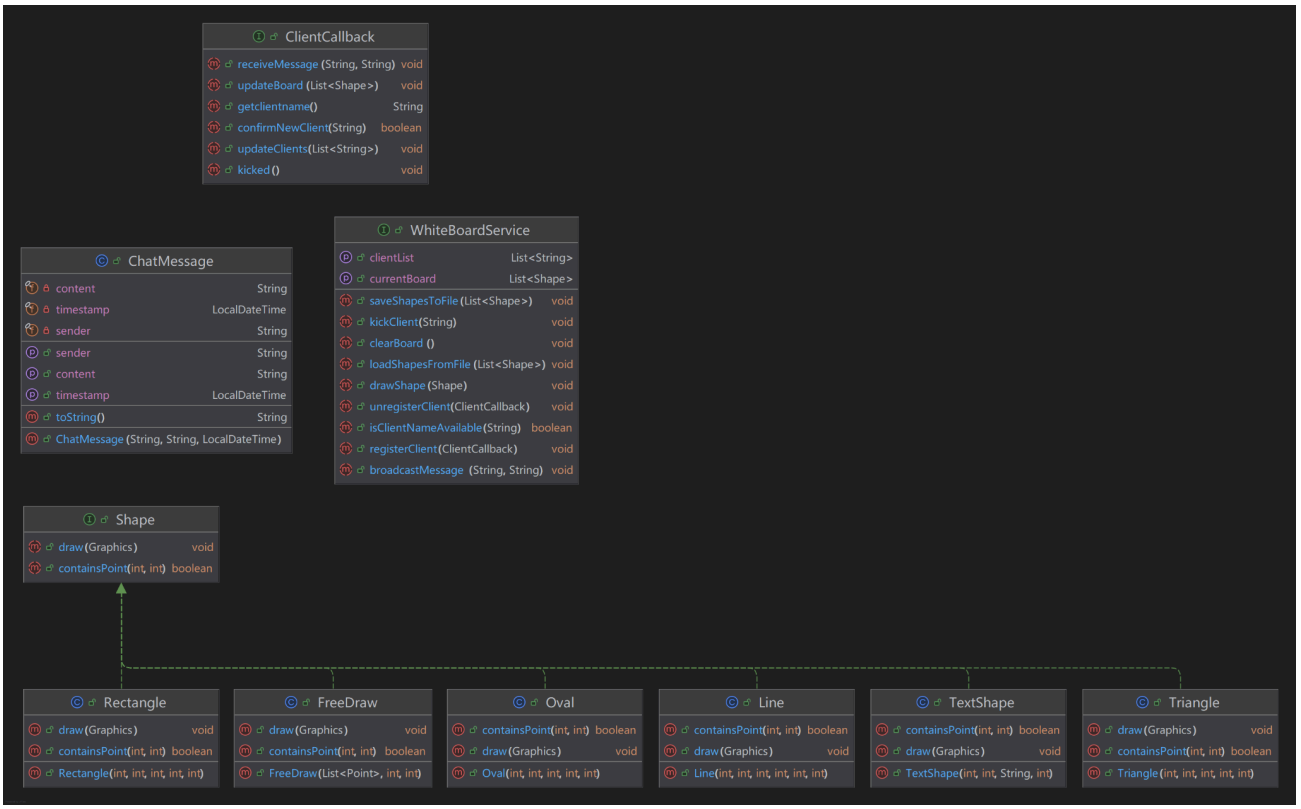
The system consists of three main components:

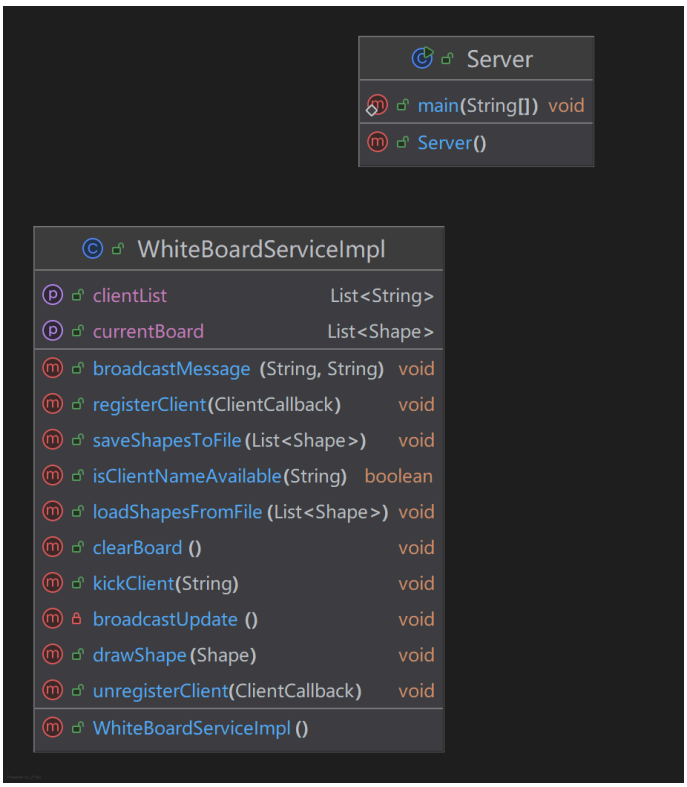
- **Server**: Handles client connections, user permissions, canvas state synchronization, and data persistence.

- **Client**: Provides a GUI for drawing, chatting, and interacting with the canvas.

- **Shared Module**: Contains RMI interfaces and shared data classes used by both server and clien

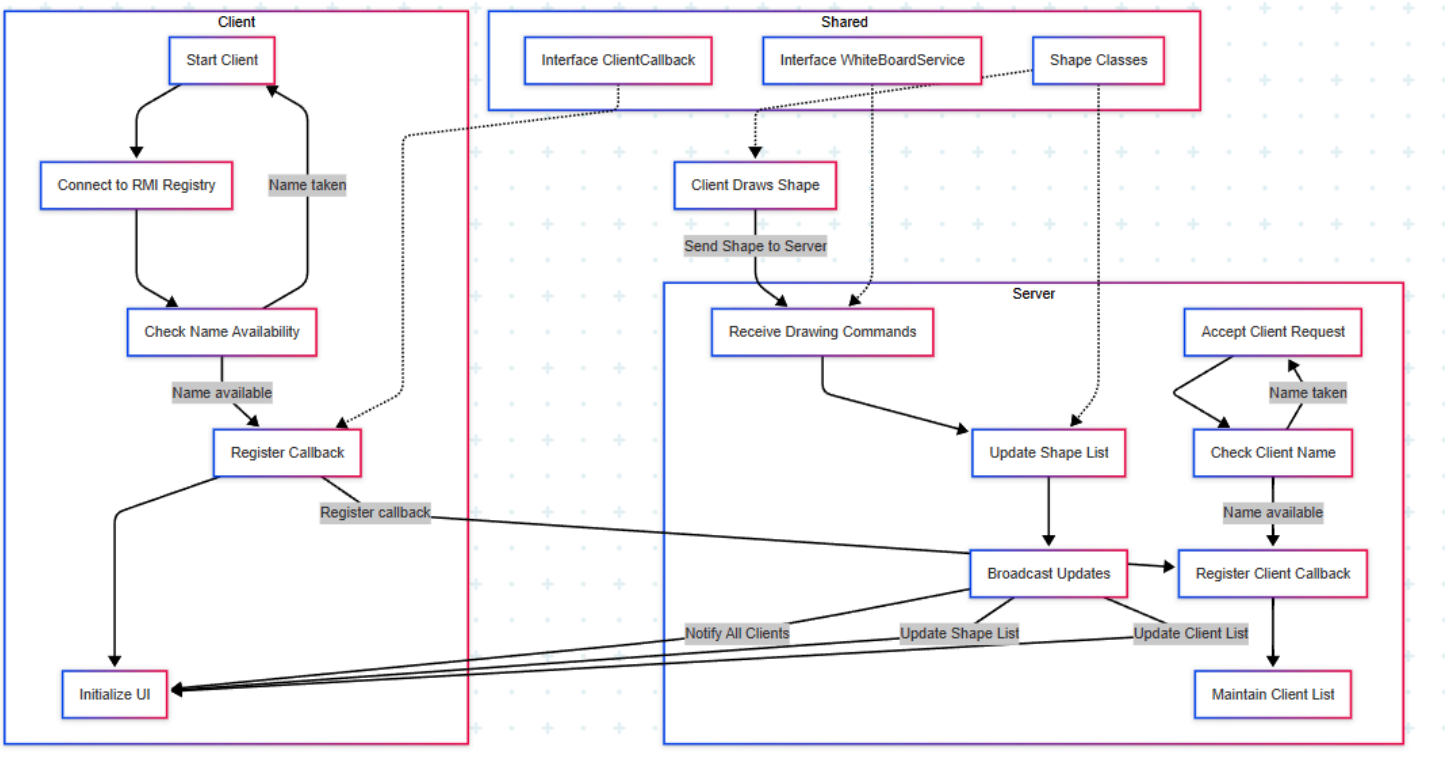

DrawingPanel
shapes List<Shape>
currentColor int
shapes List<Shape>
currentColor int
paintComponent(Graphics) void
SetSelectedShape(String) void
DrawingPanel(WhiteBoardService)
FileMenuBar
FileMenuBar(WhiteBoardService, String)
Client
canvas DrawingPanel
clientname String
clientname String
canvas DrawingPanel
setclients(List<String>) void
main(String[]) void
start() void
showMessage (String) void
Client()
ClientCallbackImpl
updateBoard (List<Shape>) void
updateClients(List<String>) void
confirmNewClient(String) boolean
kicked() void
receiveMessage (String, String) void
getclientname() String
ClientCallbackImpl (Client)


ClientCallback
receiveMessage (String, String) void
updateBoard (List<Shape>) void
getclientname() String
confirmNewClient(String) boolean
updateClients(List<String>) void
kicked() void
ChatMessage
content String
timestamp LocalDateTime
sender String
sender String
content String
timestamp LocalDateTime
toString() String
ChatMessage (String, String, LocalDateTime)
WhiteBoardService
clientList List<String>
currentBoard List<Shape>
saveShapesToFile (List<Shape>) void
kickClient(String) void
clearBoard () void
loadShapesFromFile (List<Shape>) void
drawShape (Shape) void
unregisterClient(ClientCallback) void
isClientNameAvailable(String) boolean
registerClient(ClientCallback) void
broadcastMessage (String, String) void
Shape
draw(Graphics) void
containsPoint(int, int) boolean
Rectangle
draw(Graphics) void
containsPoint(int, int) boolean
Rectangle(int, int, int, int, int)
FreeDraw
draw(Graphics) void
containsPoint(int, int) boolean
FreeDraw(List<Point>, int, int)
Oval
containsPoint(int, int) boolean
draw(Graphics) void
Oval(int, int, int, int, int)
Line
containsPoint(int, int) boolean
draw(Graphics) void
Line(int, int, int, int, int)
TextShape
containsPoint(int, int) boolean
draw(Graphics) void
TextShape(int, int, String, int)
Triangle
draw(Graphics) void
containsPoint(int, int) boolean
Triangle(int, int, int, int, int)



We use Maven structure to manage our program. The dependencies are located in pom.xml

2.2 Class Design



Shared module:

```
public interface Shape extends Serializable
```

This interface defines the base contract for all drawable shapes. It contains method declarations for shape-related operations such as draw(Graphics g)

```
public class FreeDraw implements Shape
```

This class represents a freehand drawing shape composed of a sequence of connected points. It stores the data structure for point lists and implements the logic for rendering smooth, continuous lines based on user input.

```
public class Line implements Shape
```

This class defines a straight line segment between two points. It contains data members for the start and end coordinates and implements the rendering logic to draw a line on the canvas.

```
public class Oval implements Shape
```

This class represents an ellipse or circle shape. It contains bounding box coordinates and draws an oval according to its width and height proportions.

```
public class Rectangle implements Shape
```

This class defines a rectangular shape. It maintains the position, width, and height attributes and implements the logic to render a rectangle with optional fill or border styles.

```
public class TextShape implements Shape
```

This class defines a drawable text element. It encapsulates text content, position, font, and style information and implements the rendering logic to display the text on the canvas.

```
public class Triangle implements Shape
```

This class represents an equilateral triangle shape defined by two base points. Upon construction, the third vertex is automatically calculated to form an upward-pointing equilateral triangle based on the distance and direction between the first two points.

```
public interface ClientCallback extends Remote
```

The `ClientCallback` interface defines remote methods that the server uses to communicate with individual clients in a collaborative whiteboard application. It enables real-time updates and control. Key methods include:

- **`updateBoard`** – Sends the updated whiteboard content to the client.
- **`updateClients`** – Notifies the client of the current list of connected users.
- **`getclientname`** – Retrieves the client's name.
- **`confirmNewClient`** – Confirms whether a new client can join.
- **`kicked`** – Informs the client they have been removed from the session.
- **`receiveMessage`** – Delivers chat or system messages to the client.

```
public interface WhiteBoardService extends Remote
```

The `WhiteBoardService` interface defines remote methods for managing a collaborative whiteboard in a Java RMI environment. It supports drawing and clearing shapes, handling client registration, messaging, saving/loading board content, and managing client access. Key functions include:

- **`drawShape`** / **`clearBoard`** – Modify the whiteboard content.
- **`broadcastMessage`** – Send chat messages to all clients.
- **`getCurrentBoard`** / **`getClientList`** – Retrieve the current board state and active users.
- **`saveShapesToFile`** / **`loadShapesFromFile`** – Persist and restore whiteboard drawings.
- **`kickClient`** – Remove a client from the session.
- **`registerClient`** / **`unregisterClient`** – Manage client connections.
- **`isClientNameAvailable`** – Check for name conflicts on join.

Sever module:

```
public class Server
```

The `Server` class initializes and starts the RMI-based whiteboard server. It performs the following actions:

- Parses arguments to get the host name and port number (default: `localhost`, port `1099`).
- Creates an RMI registry on the specified port.

- Binds an instance of `WhiteBoardServiceImpl` to the RMI registry with the name `"whiteboard"`.
- Launches the server GUI as a manager client using `Client.main(...)`, so the server host also participates as a whiteboard manager.

```
public class WhiteBoardServiceImpl extends
UnicastRemoteObject implements WhiteBoardService
```

The `WhiteBoardServiceImpl` class implements the `WhiteBoardService` interface and serves as the core server-side logic for a collaborative whiteboard using Java RMI. It manages connected clients, board drawings, chat, and file operations.

Key Components:

- **`List<Shape> shapes`** – Stores all shapes drawn on the board.
- **`List<ClientCallback> clients`** – References connected clients for updates.
- **`List<String> clientnames`** – Stores names of connected users.
- **`List<String> Chatmessages`** – (Declared but unused in this snippet.)

Main Functionalities:

- **Drawing & Clearing:**
  - `drawShape(...)` – Adds a new shape and updates all clients.
  - `clearBoard()` – Clears the board and broadcasts the update.
- **Client Management:**
  - `registerClient(...)` – Adds a new client after approval from the manager.
  - `unregisterClient(...)` – Removes a client and updates the client list.
  - `isClientNameAvailable(...)` – Ensures name uniqueness.
  - `kickClient(...)` – Removes a client forcibly and notifies them.
- **Communication:**
  - `broadcastMessage(...)` – Sends a chat message to all clients.
- **Persistence:**
  - `saveShapesToFile(...)` – Saves the board to a local file.
  - `loadShapesFromFile(...)` – Loads shapes from a file into the board.
- **Board Sync:**
  - `broadcastUpdate()` – Sends updated shapes and client list to all

connected clients. Automatically removes clients if unreachable.

Client module:

```
public class Client
```

The `Client` class represents the main graphical user interface (GUI) for participants in a collaborative whiteboard system, using Java RMI for communication with the remote server.

Core Responsibilities:

1. **Establish Connection with RMI Server**
   Upon launch, the client connects to the remote whiteboard server using the provided `hostName`, `port`, and `clientname`. It retrieves a reference to the `WhiteBoardService` and attempts to register itself.
2.
3. **Client Name Verification**
   Before joining, the client checks if the chosen username is already taken via `isClientNameAvailable()`. If the name is unavailable, the application exits with an error message.

4. **Remote Callback Registration**
   A `ClientCallbackImpl` instance is created and registered with the server to receive updates for the whiteboard content and client list.

5. **Drawing Interaction**
   - Users can draw various shapes (lines, rectangles, ovals, triangles, free draw, erasers, text) using the shape buttons.
   - A color palette at the top allows users to select drawing colors.
   - Drawing actions are sent to the server via `drawShape()`, and updates are broadcast to all clients.

6. **Client List and Management**

   - The left panel shows the list of connected clients.
   - If the current client is named `"manager"`, they can kick other users via a “Kick” button.
   - Kicked users are notified and disconnected using the `kicked()` callback.

7. **Chat Functionality**
   A chat panel on the right side allows users to send and receive text messages. Messages are sent through `broadcastMessage()` and displayed in the shared chat area.

8. **Graceful Exit**
   On window close, the client unregisters itself from the server using `unregisterClient()`, ensuring clean disconnection.

9. **Whiteboard Menu Bar**
   A custom `FileMenuBar` allows saving and loading of shapes from the file system (handled via the server).

GUI Layout Overview:

- **Center**: Drawing canvas (`DrawingPanel`)
- **North**: Color palette
- **South**: Shape buttons
- **West**: Client list + kick button
- **East**: Chat area
- **Top**: Menu bar (save/load options)

This class encapsulates all client-side logic and UI components needed to participate in the collaborative whiteboard session, including drawing, chatting, client management, and RMI communication.

```
public class ClientCallbackImpl extends UnicastRemoteObject
implements ClientCallback
```

The `ClientCallbackImpl` class is a concrete implementation of the `ClientCallback` remote interface, extending `UnicastRemoteObject`. It plays a critical role in enabling **two-way communication between the server and a whiteboard client** in a Java RMI-based collaborative drawing system.

Purpose

This class allows the server to **remotely update the client**, such as:

- Updating the whiteboard with new shapes.
- Notifying of new clients or removed ones.
- Broadcasting chat messages.
- Asking for client approval (in the case of a manager).

- Kicking a client from the session.

Key Features

1. **Client Association**
    - The constructor accepts a reference to a `Client` instance, allowing the callback to update the client's UI components directly (e.g., canvas, client list, chat area).

2. **Whiteboard Synchronization**
    - `updateBoard(List<Shape> shapes)`: Updates the client's local drawing canvas with the latest shape data from the server, and repaints the canvas to reflect changes in real-time.

3. **Client List Update**
    - `updateClients(List<String> clients)`: Updates the list of connected clients shown in the GUI, ensuring all participants see the current state of the session.

4. **Client Identity**
    - `getclientname()`: Returns the name of the client associated with this callback. Used by the server to identify and manage clients.

5. **Chat Message Reception**
    - `receiveMessage(String sender, String message)`: Displays a chat message in the client's chat panel when the server broadcasts a message from another user.

6. **New Client Confirmation (Manager Only)**
    - `confirmNewClient(String clientName)`: When a new user attempts to join, the server asks the manager to approve or deny the request. This method shows a confirmation dialog to the manager.

7. **Kicking Mechanism**
    - `kicked()`: If the client is removed by the manager, this method shows a warning message and forcibly exits the client application

Usage Context:

- Each client registers a unique instance of `ClientCallbackImpl` with the server upon connecting.
- The server uses this callback to maintain up-to-date client views and coordinate interactions like approvals and disconnections.

- The manager client has additional authority through `confirmNewClient`, enabling them to control participant access.

```
public class DrawingPanel extends JPanel
```

The `DrawingPanel` is a custom `JPanel` that serves as the main drawing area in a collaborative whiteboard app. It captures mouse input to let users draw shapes like lines, rectangles, ovals, triangles, freehand drawings, erasers, and text.

Key Features:

- Handles mouse events to draw selected shapes.
- Supports different tools (e.g., Line2, FreeDraw4, Erase8, Text).
- Sends new shapes to the server via RMI (`WhiteBoardService`).
- Automatically updates and repaints shapes received from the server.

This panel ensures synchronized, real-time drawing across all clients.

```
public class FileMenuBar extends JMenuBar
```

The `FileMenuBar` class extends `JMenuBar` and provides basic file operations for managing the collaborative whiteboard. It includes options to create a new board, open a saved drawing, and save the current canvas. These operations are restricted to the manager client (i.e., the user named `"manager"`).

Main Features:

- New
  Clears the current whiteboard after a confirmation dialog. Invokes `service.clearBoard()` to reset the canvas for all connected clients.

- Open
  Opens a serialized `.dat` file containing a list of `Shape` objects. After deserialization, the shape list is shared with all clients using `service.loadShapesFromFile()`.

- Save
  Saves the current canvas state (a list of `Shape` objects) to a user-specified

`.dat` file. Also calls `service.saveShapesToFile()` to store the shapes on the server side if needed.

Notes:

- Uses Java's `JFileChooser` for file selection dialogs.
- Shape objects are serialized using `ObjectOutputStream` and `ObjectInputStream`.
- Only the manager can perform these file operations, ensuring centralized control over board state.

This menu provides essential persistence support, allowing collaborative sessions to be saved and resumed.

3. Network Communication

The whiteboard application uses **Java RMI (Remote Method Invocation)** to enable real-time communication and synchronization between clients and the server. Through RMI, remote objects can invoke methods across the network as if they were local, making it ideal for collaborative applications.

Key Components:

The Server will register an RMI instance to the specified port and then start a manager client as the master client

- **WhiteBoardService (Remote Interface):**
  Defines remote methods such as `drawShape()`, `clearBoard()`, `broadcastMessage()`, `getClientList()`, `kickClient()`, etc., which are called by clients to interact with the server.

- **ClientCallback (Remote Interface):**
  Enables the server to **push updates** to each client — e.g., sending new drawings, updating the client list, broadcasting chat messages, or notifying a client that they've been kicked.

- **Client-Server Interaction:**
  - Each client looks up the `WhiteBoardService` on the RMI registry and registers itself using `registerClient()`.
  - The server stores client callbacks and pushes board updates or chat messages to all connected users using `updateBoard()` or `receiveMessage()` methods.

- The manager can call `kickClient()` remotely to remove a user.

Example Communication Flows:

- When a client draws a shape, it calls `service.drawShape(shape)`, which is sent to the server and then pushed to all clients via `updateBoard()`.
- When a message is sent in chat, the client calls `broadcastMessage(name, message)`, which the server relays to all clients using `receiveMessage()`.

This RMI-based architecture allows for **synchronous, bidirectional communication** between multiple distributed clients and a central server, maintaining consistency in both drawing content and user presence.

4. Threading Model

The whiteboard client application uses the Swing event dispatch thread (EDT) as its main threading model for handling user interface updates and event handling. Key points include:

- Swing's Single Thread Rule:
  All GUI operations such as drawing on the canvas, updating the chat area, and responding to button clicks happen on the Event Dispatch Thread (EDT) to ensure thread safety of Swing components.

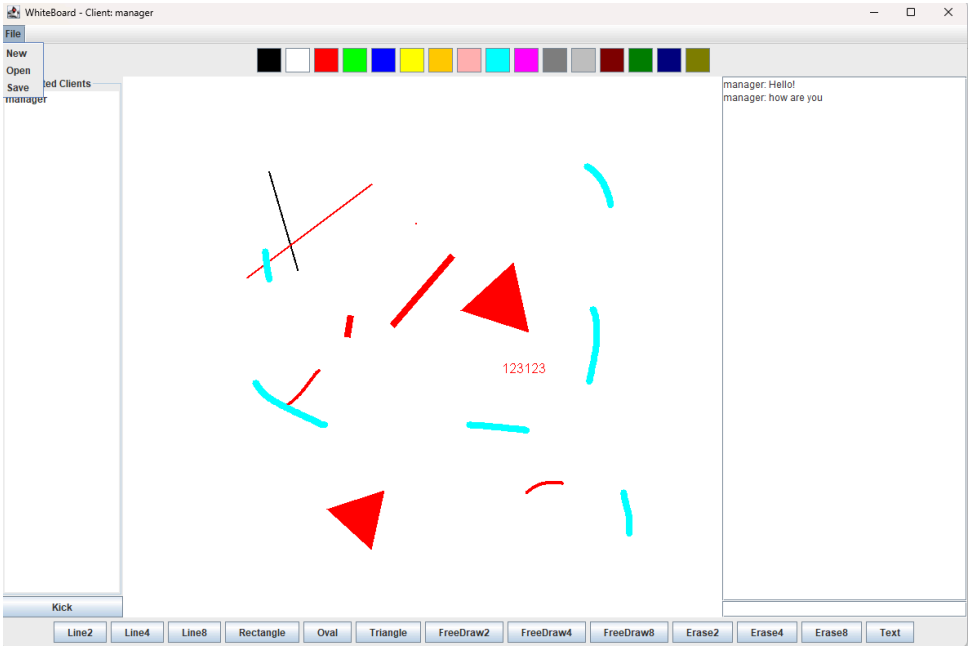
- Background Network Calls:
  Remote calls to the RMI server (e.g., drawing shapes, broadcasting messages, fetching client lists) occur on separate threads initiated by Swing event handlers, but network-related updates that affect the UI (like receiving new board data or chat messages) are marshaled back onto the EDT using `SwingUtilities.invokeLater()`. This prevents concurrency issues by ensuring UI updates run safely on the EDT.

- Callback Handling:
  The `ClientCallbackImpl` methods invoked remotely by the server typically update the GUI. These methods wrap their UI operations in `SwingUtilities.invokeLater()` to ensure updates occur on the EDT.

- Mouse Events:
  Mouse listeners and motion listeners on the drawing panel process user input on the EDT, collecting drawing points and sending remote commands accordingly.

5. Client-Side Design

6.1 GUI Overview:



As shown in the figure, our GUI design is controlled by the options of multiple panels. At the top is color, at the bottom is shape, and the artboard is in the center. Users can chat on the left. This information is synchronized across all clients. The left side shows the clients we are connecting to, our moderator is the manager by default, the manager can select any client and kick out, and can save or load the current artboard. If you're not a manager, you can't use these features。

7. Error handling

1.RemoteException Handling

Context: All remote method invocations on the `WhiteBoardService` interface (e.g., drawing shapes, clearing the board, kicking clients) may throw `RemoteException` due to possible network or communication failures.

Implementation: These exceptions are caught using try-catch blocks around every remote call. Upon catching, the application either:

- Logs the stack trace for debugging purposes.
- Displays error dialogs to inform the user.
- Or throws a runtime exception to propagate critical failures.

2. File I/O Exception Handling

- Context: The client supports loading and saving drawing data via file streams which can fail due to missing files, permission issues, or corrupt data.
- Implementation: File reading and writing use `try-with-resources` to safely manage streams, and catch both `IOException` and `ClassNotFoundException` during deserialization.
- Errors are logged to the console for diagnosis, but do not crash the application.

3. User Input Validation and Confirmation

- Context: User dialogs for text input and action confirmations may return invalid or null results if the user cancels or enters nothing.
- Implementation: The application checks user inputs before proceeding (e.g., ensuring text is not null or empty), and confirms destructive actions like clearing the canvas with confirmation dialogs.
- This prevents invalid state changes or unintended data submission.

4. Application Startup Error Reporting

- Context: The initial connection to the RMI service may fail due to server unavailability or network issues.
- Implementation: Exceptions during startup are caught and displayed in a message dialog, informing the user that the client failed to start.

8. Critical Analysis

1. Architecture

- The code is organized with clear separation between UI, network, and logic.
- Role-based features (e.g., manager rights) are well implemented.
- However, UI and network code are tightly connected, making future changes harder.

2. Network Communication (RMI)

- RMI enables easy remote calls and callbacks for real-time updates.
- The client can approve new users, adding some security.
- But there is little handling for network failures or retries.
- Concurrency control for shared data is missing, which may cause errors when multiple updates happen simultaneously.
- The manager's client quit will not lead other client quit safely.

3. User Interface

- The drawing panel and menus are intuitive and support many tools.
- UI updates are correctly done on the Swing event thread to avoid bugs.
- But network calls happen on the UI thread, which might freeze the interface during delays.
- Frequent modal dialogs can interrupt user experience.

4. Threading

- UI updates use proper threading (EDT).
- However, there is no explicit locking for shared data, risking race conditions.
- Network calls are mostly synchronous and block the UI.

5. Error Handling

- Common exceptions (network, IO) are caught and some user messages are shown.
- Resource cleanup (e.g., file streams) is handled well.
- Error handling is inconsistent and lacks retry or recovery strategies.