The University of Melbourne
School of Computing and Information Systems
COMP90015 Distributed Systems

Assignment 2 - Distributed Shared Whiteboard

Marco Marasco 834482

Introduction

This report will outline and discuss the implementation of a Distributed Shared Whiteboard application implemented in Java following a client-server model. This application was designed to allow concurrent clients to draw and edit a shared whiteboard, with any changes made available to all clients in real time. Communication between the client and server is via Java Remote Method Invocation (RMI). The application also has a myriad of additional features and failure handling capabilities in its design, ensuring errors occurring on both the client and server do not impact the reliability and user experience of the application.

System Architecture

When designing the overall architectural approach for the system design, the obvious approach was to follow a client-server model, using a single server instance. The system required a single management user to control the application, a client-server model most appropriately fitted the problem domain. Furthermore, the scope of the application didn't require multiple servers to assist in distributing the application load, as the application operations were relatively computationally cheap, but from an economical perspective, the added utility of additional servers did not justify the increased system complexity. As such, a many-to-one client-server architecture was used for this application.

Below shows an architectural diagram of the system:

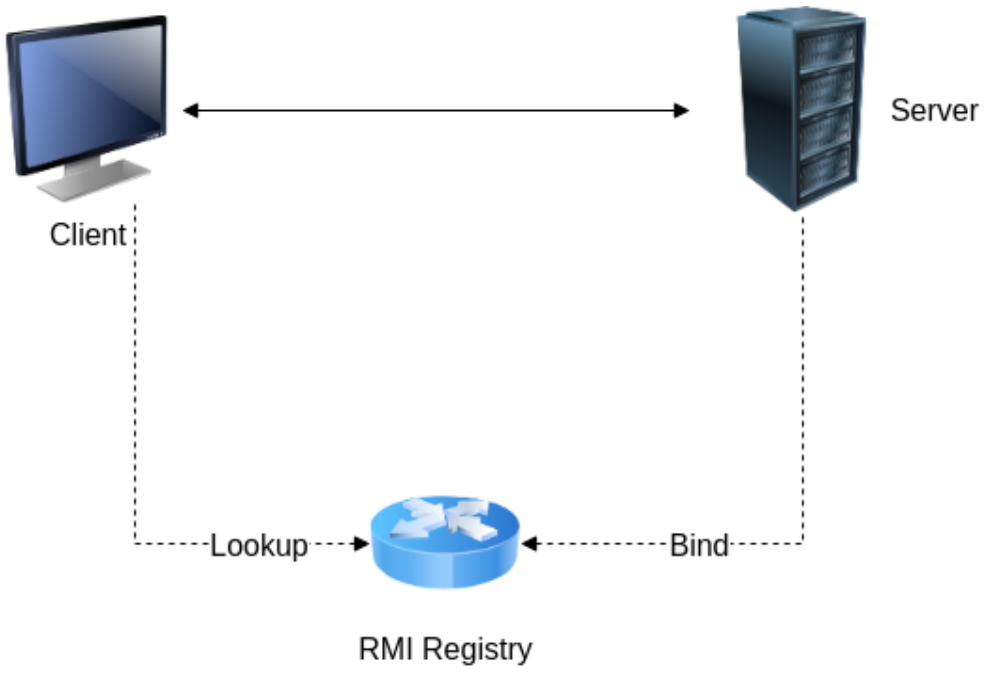


Figure 1: System Architecture

System Components

Server

As previously stated, this application used Java RMI as the middleware for allowing collaborative editing of a shared whiteboard. RMI was chosen for client-server communication as it:

- Provides greater transparency compared to using Sockets for communication.
- Natively implements mechanisms to handle messaging across the TCP/IP stack, concurrency, and marshalling objects.

When the application is started, the server component creates and hosts a RMI registry at runtime on an input port, and by binding a server management object to the registry, allows for clients to join the system application. After creation of the RMI registry and binding of the server manager, the server displays its own instance of the shared whiteboard, as well as a list of all connected users (only itself initially). To increase control over the application clients, the server was designed to be able to accept or reject join requests by clients. Careful consideration was made into whether multiple clients could join the server with the same username. As the list of currently connected clients displayed each username, having duplicate usernames was deemed to be inappropriate as users could not be identified individually. As such, in the event a user attempts to join with a duplicate username, they are immediately rejected from the server, and notified the username is already in use.

The server also has several management capabilities to provide a professionally acceptable experience. Similar to the functionality of Microsoft's Word document editing application, the server is able to create new whiteboards, save a current whiteboard, open a saved whiteboard, or save a current whiteboard under a new file (save as). To increase the completeness of the user experience, the server also has measures to prompt the user to save any changes if the request to perform an action that will cause loss of unsaved changes. An example scenario is:

1. Server creates a new whiteboard.
2. Shapes are drawn on the whiteboard.
3. Server clicks **load** button (without having yet clicked save).
4. Server is prompted to save the current changes (in this scenario, to a new file).

This added measure to prompt the user to save the current changes ensures the risk of accidentally losing whiteboard shapes is heavily mitigated.

To increase the control over the application clients, the server also has the ability to remove clients at will. If a client is removed, they are notified by the server, and their whiteboard is disconnected and their running client application is terminated. The server then updates the list of currently connected users to reflect the changes. If a client disconnects, the server detects this via no responses from periodical ping messages to its current clients. When the server closes their application. In the event the server closes the whiteboard application, first they are prompted to save the changes, then all connected clients are notified of the closure and are shutdown. This ensures a seamless experience for clients of the server termination, rather than suddenly not being able to draw on the shared whiteboard.

Client

The client class was designed to be simple, yet sufficiently robust and responsive for this application. Each client defines its own username, as well as a RMI server address and port to connect to, allowing multiple clients on a single machine to connect to multiple whiteboard servers. Client's are appropriately notified if they have input incorrect server details, or if they have been accepted/rejected from the server.

Each client utilises two threads for the application. One thread is concerned with drawing on its whiteboard and sending its shapes to the server, the other is concerned with accepting and handling messages from the server. This approach was used to increase the concurrency of the system, as it allowed a client to draw on the whiteboard whilst concurrently seeing other clients' drawings being made in real time. Clients also have a list of all currently connected users in their whiteboard GUI, allowing them to know who they are currently collaborating with.

Whiteboard GUI

For any interactive application, careful and considered design of the graphical user interface is key to the success of the application. Even if the underlying application logic is extremely efficient and responsive, a bad GUI decreases the user experience, and essentially renders the application undesirable to use. The whiteboard GUI is enclosed within a Java Swing JPanel, and possesses a large whiteboard, a detachable toolbox, and a list of all clients currently connected to the whiteboard server. By allowing the toolbox to be detached, clients are able to make the decision of whether they want a more immersive drawing experience, or have all tools at a closer proximity to the board. The whiteboard GUI implements several features to provide a positive user experience with this, as outined below:

- Multiple shapes can be drawn (Rectangle, Ellipse, Line, Round Rectangle, Star)
- Completely customisable colour palette (allowing for any colour to be selected).
- Able to choose if shapes are filled, or just outlines.
- Eraser option to remove shapes.
- Text input.
- Button to clear the whiteboard.

Each of these features allows for a fully interactive, highly customisable shared whiteboard experience.

Storage

As the server was required to be able to save and load previously constructed whiteboard, during the design phase several methods were considered on how to provide this persistence between runtime executions. The first option considered was encoding each whiteboard into a JSON/text file that could be loaded later. The encoding method was hypothesised to encode each shape into a text form containing its metadata (size, shape, position etc). The main advantage of this method was that the saved file would be human readable, and could be manually created or edited offline. However, this was paled in comparison to the

computational overhead of encoding each of the objects, as converting each object to a string representation was expensive as the number of shapes grew.

The next two options considered both involved serialisation of the objects in the whiteboard (further description in the communication section). These two options were to save the whiteboard state to a database, or to save it to a file. The database option allowed for efficient recording of the whiteboard state (compared to the string encoding method), but came at a significant cost. The computational overhead of connecting to the database, then writing to disk was unnecessary for the application domain. Further, this method was not as fault tolerant nor persistent as serialising the whiteboard state to a direct file. By using a database, the system would have to ensure the database was operational, which is more likely to be crashed compared to the local machine's file system (and if the local file system had crashed, it is unlikely the application will behave correctly in any case). Steps could have been taken to increase the fault tolerance and safety of the database, such as using a distributed database cluster, and using replication to ensure redundancy, but this was too complex for the application. It was likely that maintaining the database would become more computationally expensive than the whiteboard application itself. As such, using the local machine's file system to save whiteboard states into separate files was deemed to be the most efficient and persistent method of storing whiteboard states for this application.

Concurrency

As this is a distributed application, achieving a high level of concurrency during the design phase was deemed to be of the utmost importance.

At the server side of the application, two activities can simultaneously occur:

- The server manager is drawing on their whiteboard.
- The server is receiving/sending messages to clients.

To combat this, the server uses two threads, one for each scenario above. This allows for the server user to not be impacted by their local application's responsibility of hosting the shared whiteboard, as the server can be drawing in one thread, and handling messages in another.

A similar approach is taken to handle concurrency in the client side, where:

- The client is drawing on their whiteboard.
- The client is receiving messages from the server.

By abstracting each of these responsibilities to separate threads, the clients are able to draw and see drawings being made all in realtime, without any impact from the distributed nature of the application.

Communication

Whilst RMI was established as the middleware for communication, significant thought went into how to design the communication protocols for this application. Particularly, when were messages were sent, who were they sent to, how they were received.

When approaching the communication protocol design, the greatest challenge was the many-to-one client-server model. The questions raised where:

- How do clients get notified of changes in the server?
- Who do clients communicate with? All other clients or just the server?

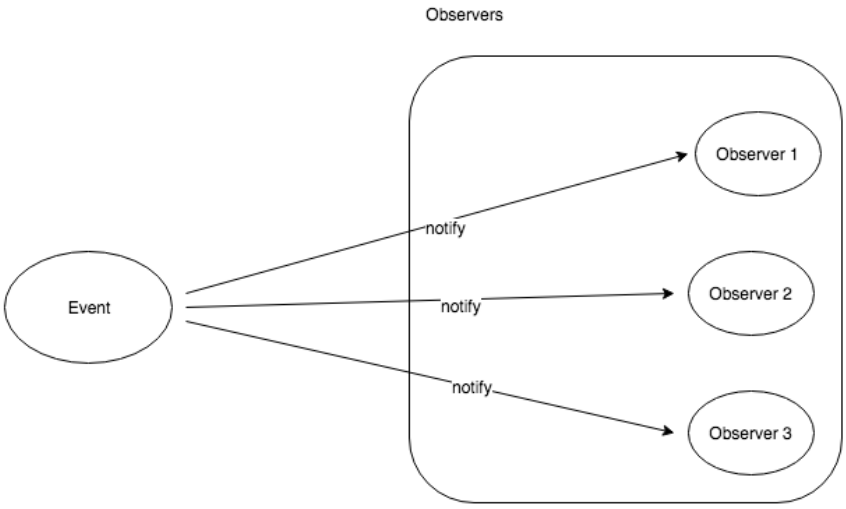


Figure 2: Observer Design Pattern

A common approach to the first problem is to use the observer design pattern. This pattern contains an object, called the subject (server), which maintains a list of its dependents, called observers (clients), and notifies them automatically of any state changes, usually by calling one of their methods. This pattern allowed for efficient, simple notifications to clients of whiteboard state changes, and was used as the pattern for distributing messages. A solution to the second problem flowed from the solution to the first. By having a well defined method to distribute messages from the server to the clients, if a client wanted to submit a shape to all clients, it would send a message to the server to then be distributed to the other users.

Message types in this application can be defined into whiteboard, and non-whiteboard related. Whiteboard messages are concerned with the whiteboard state, add shapes to the board etc, whereas non-whiteboard messages are either messages for a client joining or being kicked from the server, and the server closing.

Non-whiteboard messages follow a simple protocols. Join requests pass enumerated Java types signifying the join request result, and the kick/close messages are performed by the server invoking a client-side method to handle the process. When new clients join, the server invokes a method in each client to update their list of usernames with the new username.

The protocols for whiteboard related messages have greater complexity, and make strong use of the observer pattern. To be able to distribute the whiteboards state across client,

the very nature of what the whiteboard state was had to be well defined. The whiteboard was being sequentially rendered from a local array of shapes, where each shape contained the necessary data about the shape (shape, size, colour, position), naturally, the whiteboard state was classified as an array of shapes, with the agreed state being the array stored at the server. By rendering the shapes sequentially from the array, shapes were always guaranteed to be above/below other shapes in the correct order they were drawn, i.e. shapes drawn on top of other shapes appeared that way for all clients.

Drawing/Receiving a Shape
When a client draws a shape, the client (using a separate message controller class) sends the shape object to the server to be added the server's local array of shapes. The server adds the shape to its own list, and then using the observer pattern, notifies all clients of a new shape to add to their list. Each client adds the shape, and re-renders their whiteboard to reflect the changes.

Client joins existing whiteboard
When a client joins an existing whiteboard, after it has been registered with the server and accepted, it requests the whiteboard state (array of shapes) from the server, and then renders its whiteboard accordingly.

A new whiteboard is created/loaded
In the event that the whiteboard server manages creates a new, or loads an existing whiteboard, each client needs its whiteboard state to match the new whiteboard state. This particular scenario differs from the previous two, as this scenario is invoked by the server, not the client. To ensure that all clients update their board in real-time to the newly created/loaded whiteboard, when the server creates/loads a whiteboard, it invokes a method in all of its client that forces them to refresh their whiteboard state with the new whiteboard state at the server. This method is similar to how the server enforces users to have an up to date list of usernames.

Diagrams

Screenshots of the GUI, as well as the class and key interaction diagrams are presented in the pages after the conclusion of this report.

New Innovations

Below outlines some of the new innovations added for this application, that extend its functionality and user experience beyond the minimum requirements:

- User Friendly GUI
  - This application employs a User Friendly GUI to ensure smooth and safe operation during runtime. By providing a pleasant interface, the application is likely to be more fully utilised and explored by clients.

- Runtime Data
  - The addition of a timer for the server manager to know how long they have been running the whiteboard server, and displaying the relevant server address provides an additional element of knowledge over their application.
- Input data checking
  - The data input to the program via the commandline is checked if it is valid, resulting in unnecessary attempts to start/connect to servers are removed.
- Graphical Local File System Navigator
  - By providing server managers a GUI to navigate through their local file system, they are able to open previously existing whiteboards stored in any directory they want easily, without having to remember an extensive file path.
- Custom Registry Creation
  - At each runtime, an RMI Registry is created at a specific port number input by the server manager. In doing so, only one whiteboard server can operate on that particular port. This allows for a single machine to be able to host multiple whiteboards. A scenario where this would be a beneficial feature is a Distributed Systems tutorial, where the tutor hosts multiple boards, and the class is divided into smaller groups that each collaborate on a separate shared whiteboard.
- Save Changes Checks
  - Whenever the server performs an action that will change the whiteboard state using the server-privilege operations (new/load), the system checks if any unsaved changes have been made, and prompts the server manager to save these unsaved changes. This minimises the risk of data loss in the system.
- Additional Shapes & Colours
  - By adding in additional shapes and a full colour spectrum, the level of potential creativity in the whiteboard is vastly increased, allowing for a more immersive experience, and broadens the potential of what can be created on the shared whiteboard.
- Eraser & Clear Whiteboard
  - Allowing whiteboard users to have the ability to erase or clear the whiteboard further extends the functionality of the whiteboard, particularly in the context of removing mistakes. The clear functionality is especially useful in the Distributed Systems tutorial example, as it allows a tutor to clear the whiteboard easily, and write new content for students.

Future Work

Although this implementation provides a robust application, there is room for future development. Firstly, whilst Java's Swing library is an excellent choice for GUI design in Java, it does lack behind libraries provided in other libraries such as JavaScript. As such, future work could be made into changing the GUI to operate with a different service, but still maintain Java's RMI for communication between the client and servers. Secondly, the system is has significant security risks due to the un-encrypted messages communicated. For larger scale scenarios, it would be worth utilising some method of encryption to protect the privacy of the whiteboard.

Conclusion

This report outlines a strong implementation of a Distributed Shared Whiteboard implemented using Java. Each of the components were designed to be scalable, extendable, and able to appropriately handle faults in the system. The implemented GUIs provided a safer, immersive user experience, with great capacity for artistic creativity. As mentioned, further improvements could be made into the graphical capabilities and safety of the system.

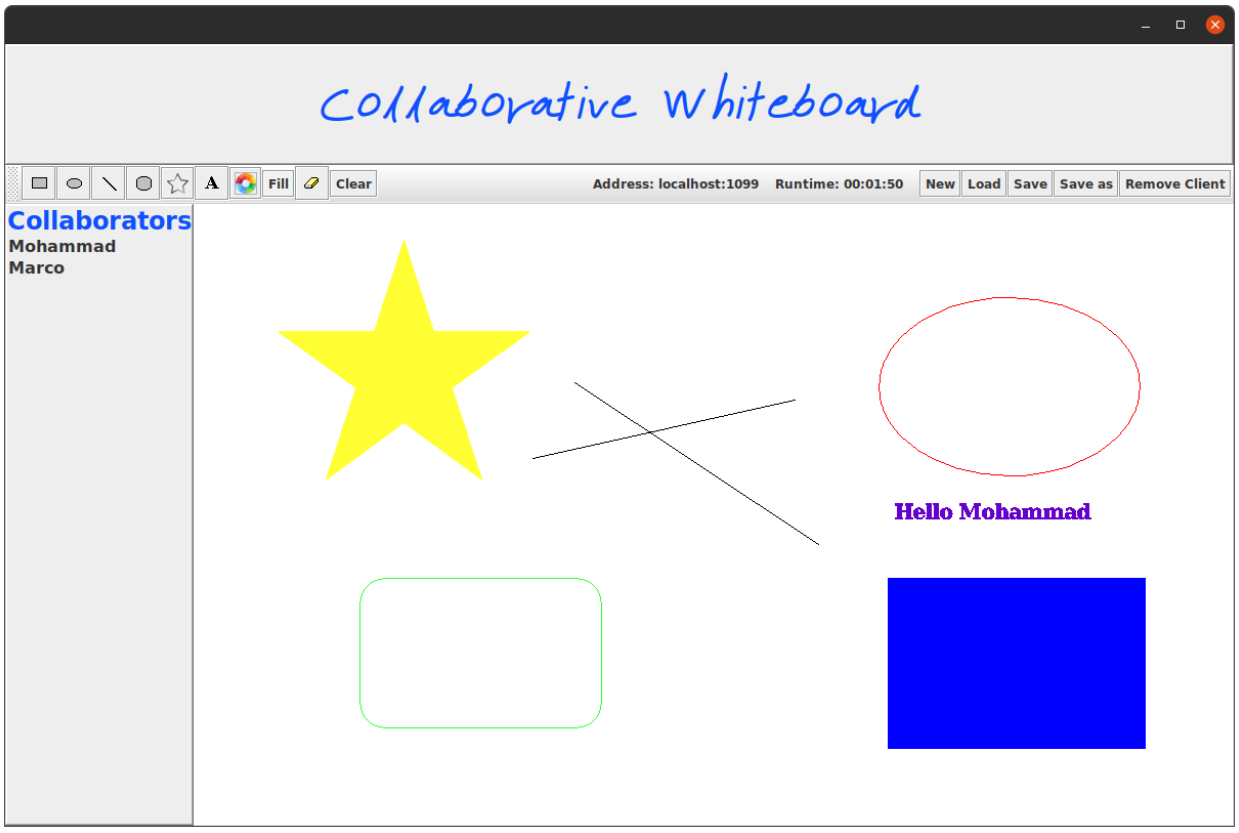


Figure 3: Whiteboard Manager GUI

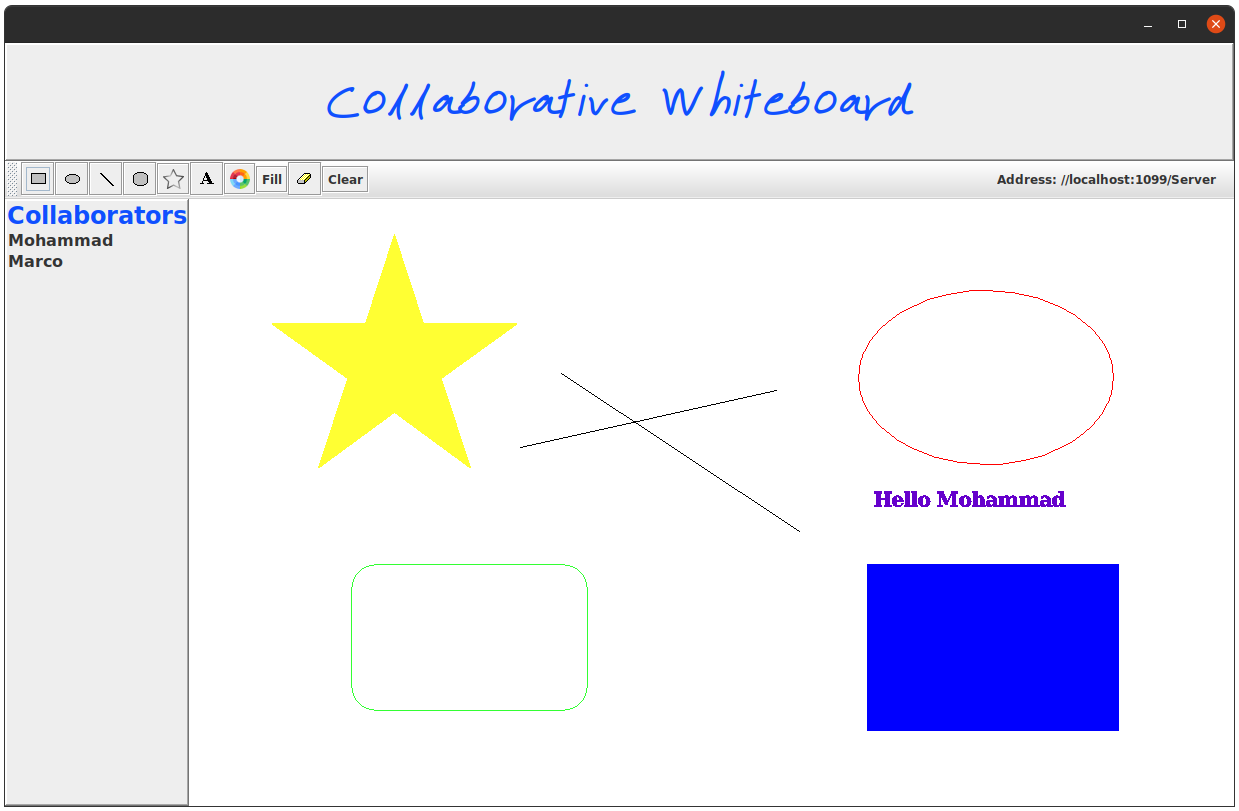
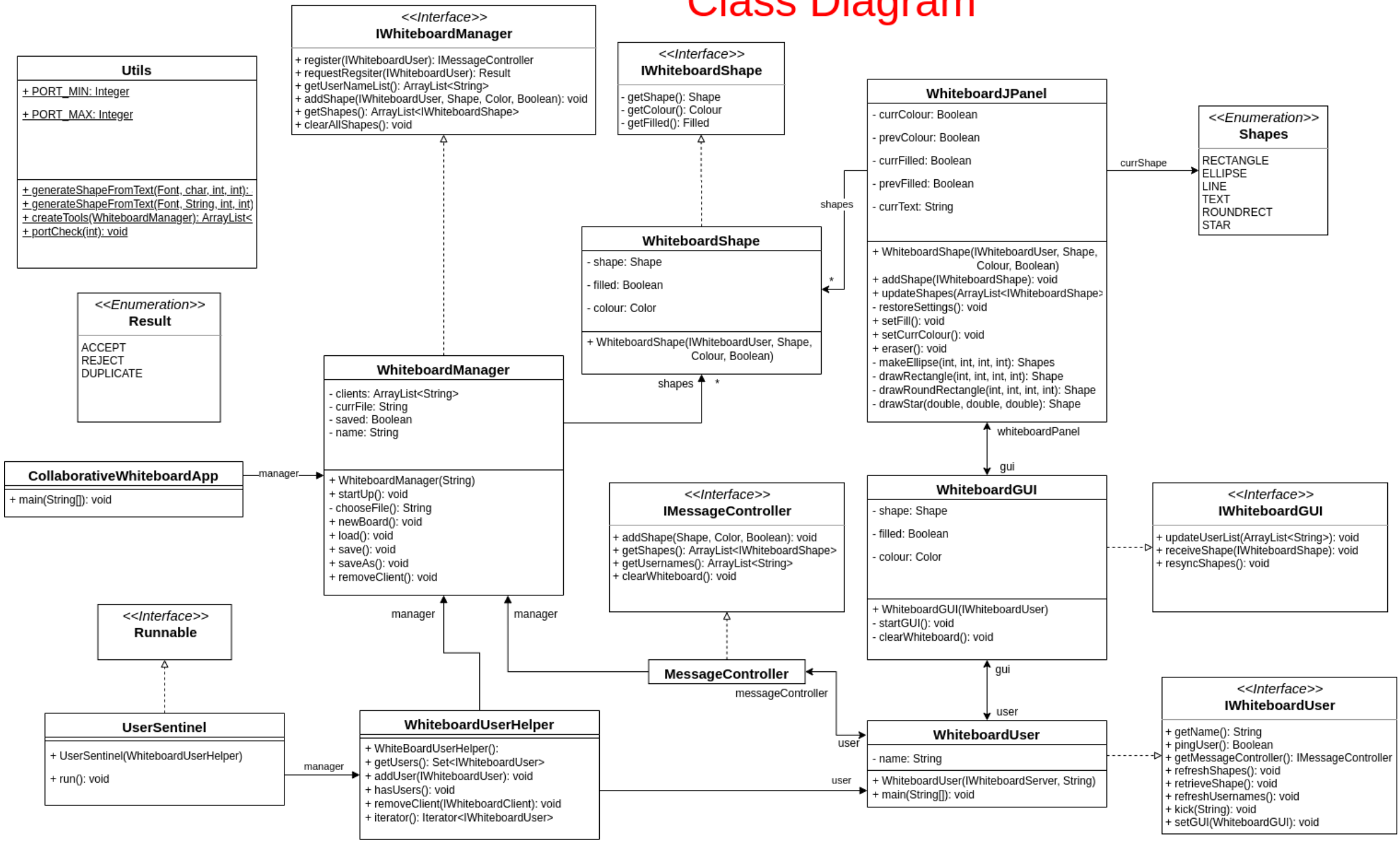
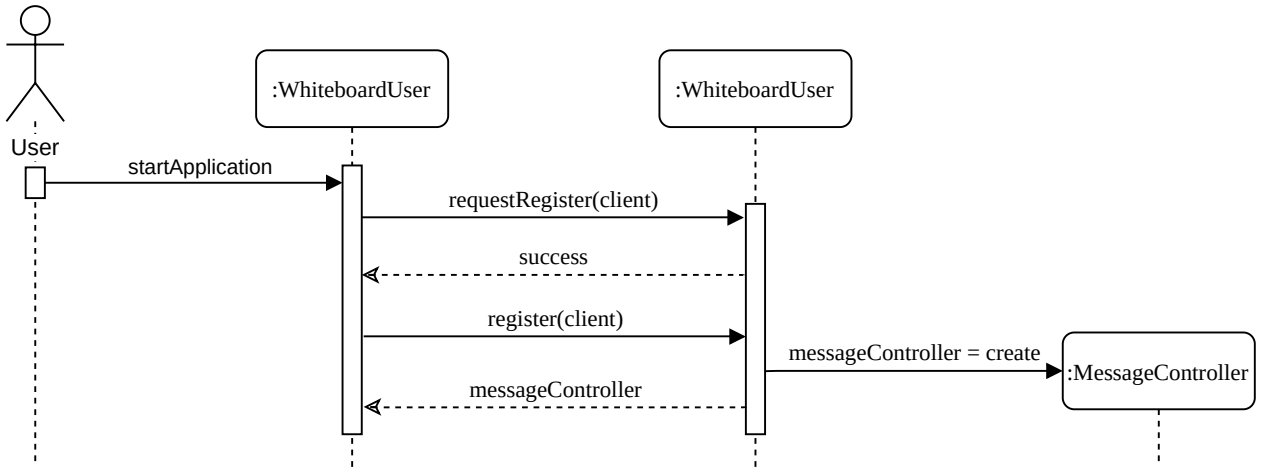


Figure 4: Whiteboard User GUI

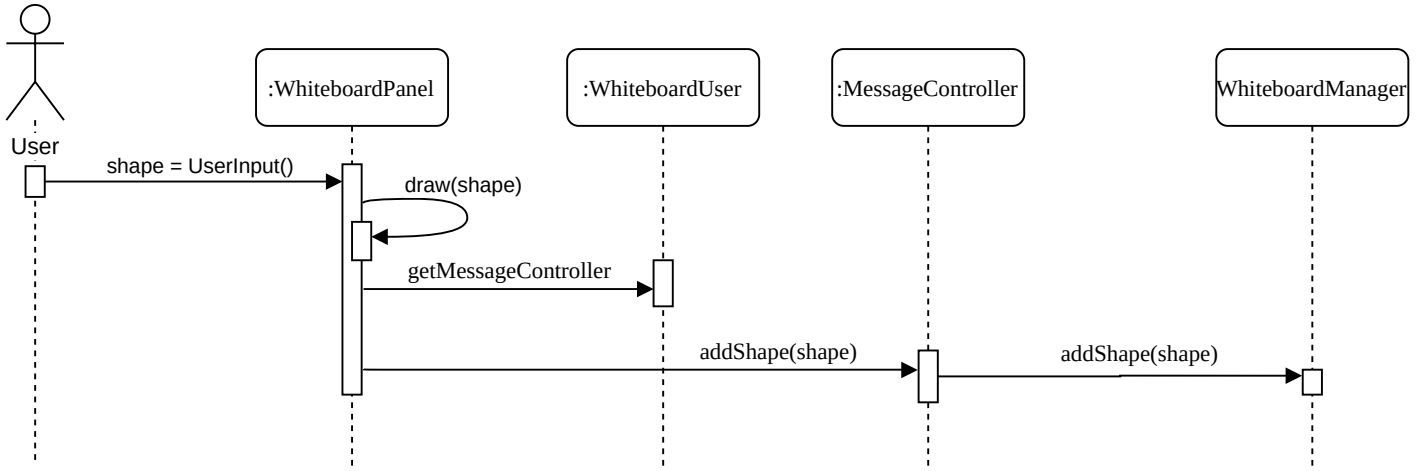
Class Diagram



User Registration



User Draw Shape



Server distribute shape