

meet.me : UX

*A Design Specification Document for the
course CS5617 (Software Engineering)*

Design Specification Document

by

UX Team

Under the supervision of
Ramaswamy Krishnan Chittur



INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

Department of Computer Science and Engineering
Kerala, India - 678557
October 2021

Contents

1 Overview	4
1.1 Design Pattern	4
2 Objectives	4
2.1 Phase - 1	4
2.2 Phase - 2	5
3 Team	6
4 Authentication(Server)	6
4.1 Members	6
4.2 Objective	6
4.3 Roles	6
4.4 Interfaces and classes	7
4.5 UML Diagram	7
4.6 Activity Diagram	7
4.7 Design	8
5 Authentication(Client)	8
5.1 Members	9
5.2 Objective	9
5.3 Interfaces and Classes	9
5.4 UML Diagram	9
5.5 Activity Diagram	10
5.6 Design	10
6 Home Page	10
6.1 Members	11
6.2 Objective	11
6.3 Interfaces and Classes	11
6.4 UML Diagram	11
6.5 Activity Diagram	12
6.6 Design	12
7 WhiteBoard UX	12
7.1 Members	12
7.2 Objective	13
7.3 Roles	13
7.4 Interfaces	13
7.5 UML Diagram	15
7.6 Activity Diagram	15
7.7 Design	16

8	ScreenShare UX	16
8.1	Members	16
8.2	Objective	16
8.3	Interfaces and classes	16
8.4	UML Diagram	17
8.5	Activity Diagram	18
8.6	Design	18
9	Chat UX	19
9.1	Members	19
9.2	Objective	19
9.3	Roles	19
9.4	Interfaces and classes	20
9.5	UML Diagram	20
9.6	Activity Diagrams	21
9.7	Design	22
10	Layout & Management	23
10.1	Members	23
10.2	Objective	23
10.3	Design	23
11	DashBoard UX	27
11.1	Members	27
11.2	Objective	27
11.3	Interfaces	27
11.4	UML Diagram	28
11.5	Activity Diagram	29
11.6	Design	29
12	Module Level Interaction	31

1 Overview

The UX or User Experience module is an important module that is responsible for all aspects pertaining to the end-user experience of the application and is tasked with providing a user-friendly interface that allows users to interact with the backend seamlessly. In our application, UX interacts with the *Content* module for providing textual chat-related features, *WhiteBoard&Screenshare* module for providing an interactive whiteboard and screen sharing feature, and the *Dashboard* for authentication and summary feature.

1.1 Design Pattern

For this application, the UX module follows a Model View-View Modal (MVVM) Design Pattern. MVVM helps provide a rich UI, easier testability of features, code reusability, and complex data binding. It helps to improve the separation of the core logic for various features and UI without any direct communication between each other. The design and development can work together without interrupting each other.

- **View** - This is the user interface and is primarily written in XAML. All the components that need to be rendered on the screen so that a user can see it are done here.
- **Data Model** - This is the backend and it is where the algorithms that support the various features reside.
- **View Model** - This is a non-visual and is responsible for connecting the View and the Data Model.

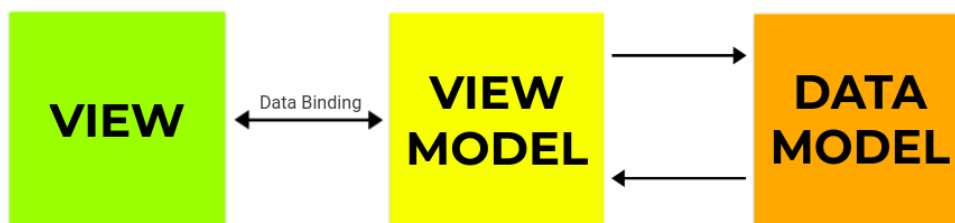


Figure 1.1: MVVM Design Pattern

2 Objectives

2.1 Phase - 1

The following features have been finalised for phase-1 after extensive deliberation with corresponding modules and have been described in detail in this design specification document.

1. Displaying dynamic list of users who are part of the session.
2. An Interactive WhiteBoard with features to select, draw, rotate and modify pre-defined shapes, eraser tool, locking of whiteboard, show shape ownership etc.
3. A Chat feature that allows users to send and receive messages and files, promote messages and reply to them.
4. A ScreenShare feature that allows users to share the contents on their screen to other users.
5. A Summary feature that allows users to dynamically see the summary of the conversations that take place in the chat. It also displays various other useful statistics.
6. A two page UI which users to use and interact with all these features.
7. Authentication page that verifies a user who wants to enter a meeting based on IP address and password.

2.2 Phase - 2

The following features are ambitious features we would like to support but heavily depend on constraints in time, complexity and difficulty the data models may face in realizing them.

1. Providing a **reply in thread option** to the users, so that the replies are connected to the original message. The user must select a thread to open it in the original conversation.
2. Editing an already sent Message.
3. Private Messaging.
4. Supporting Emojis for chat.
5. A email id based authentication.
6. Free hand drawing would have quality of life improvements, along with checkpointing of whiteboard state.
7. Further inferencing insights from summary text obtained like suggesting the list of topics discussed during the session.
8. Realtime implementation of the analytics such as displaying interactive graphs, etc.
9. Including more data and insights for analytics in the dashboard view. Thus, improving the final layout to make it as user friendly as possible.

3 Team

- **K M Raswanth**, 111801056, Team Lead
- **Sai Vipul Mohan**, 111801045
- **Irene Casmir**, 111801017
- **P S Harikrishnan**, 111801028
- **Aniket Singh Rajpoot**, 111801051
- **Arpan Tripathi**, 111801005
- **Vinay Kumar**, 111801034
- **Suchitra Yechuri**, 111801043
- **Mitul Kataria**, 111801025

4 Authentication(Server)

4.1 Members

- Sai Vipul Mohan - 111801045
- Vinay Kumar - 111801034

4.2 Objective

- Designing a Command Line Interface(CLI) to facilitate starting the meeting and terminating all the meetings from the server side.
- Implementing **startMeet** command to create a new meeting and get the ip address and port number of the meeting from Session Manager.
- Implementing **endMeet** listener to listen to meeting ending event and terminate the CLI session.

4.3 Roles

- **Vinay Kumar:**
 - Designing Command Line Interface window.
 - Creating a meeting from CLI window using **startMeet** command.
- **Sai Vipul Mohan:**
 - Listening for the end meet event from CLI window and terminating the meeting on receiving such event.

4.4 Interfaces and classes

No Interfaces or Classes will be implemented by Authentication(Server).

4.5 UML Diagram

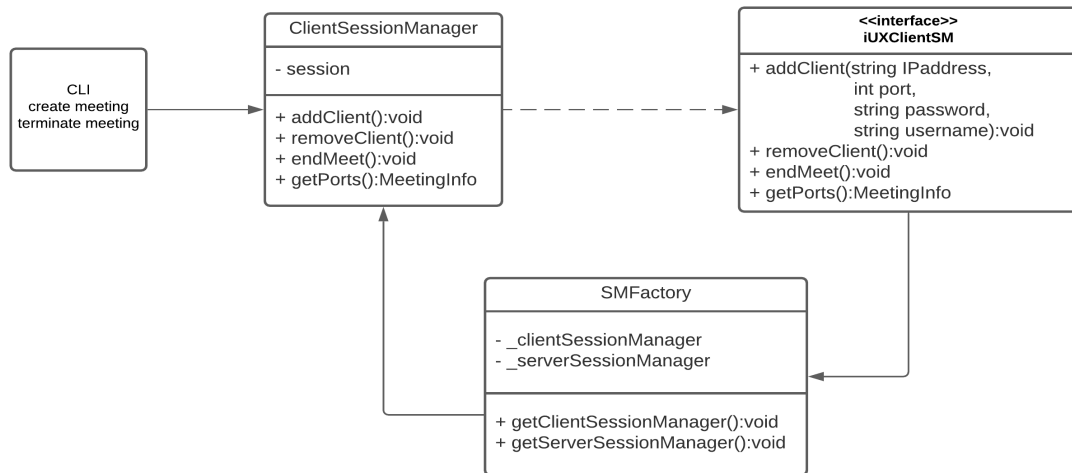


Figure 4.1: UML diagram for Authentication(Server)

4.6 Activity Diagram

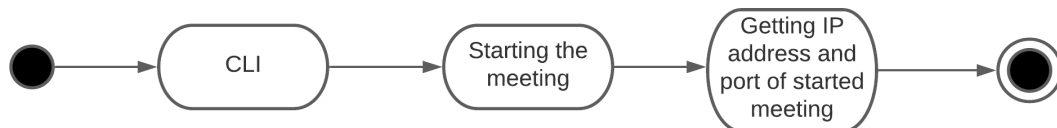


Figure 4.2: Starting the Meeting using CLI.

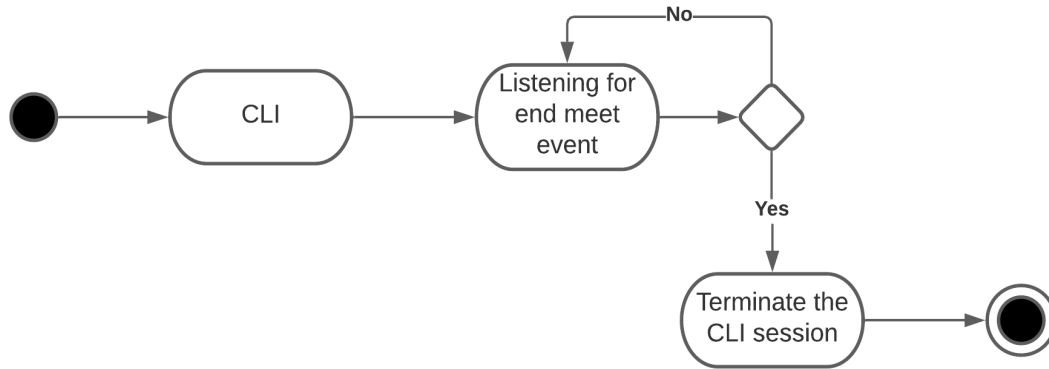


Figure 4.3: Terminating the CLI session.

4.7 Design

The Command Line Interface (CLI) of the UX module sits on top of all other modules. The application follows as server-client module. When a host wants to start a meeting he/she must first obtain the IP address and the Port Number which can then be shared by the host to others whom he/she would like to invite into the room. The other members can enter these credentials into the Authentication(client) to enter that particular session/room.

Meeting is instantiated by the UX module using the CLI and the meeting credentials like ip address and port number on which the meeting is running in the current server are obtained and if required passed to other components of the chat module for rendering onto the UX. To instantiate the meeting, **startMeet** command (on CLI) is provided which in turn calls the **getPorts** method of the **session management** module. The **getPorts** function returns an object containing ip address and port number of the meeting.

The Host's userid is reserved as 1 and if he/she clicks on the leave meeting button present on the homepage then the **endMeet()** method (in **ClientSessionManager**) is called. When this happens the session manager closes all the meetings and informs the CLI and it is closed.

5 Authentication(Client)

Develop the UX for session authentication to let a user join the room of interest.

5.1 Members

Irene Casmir - 111801017

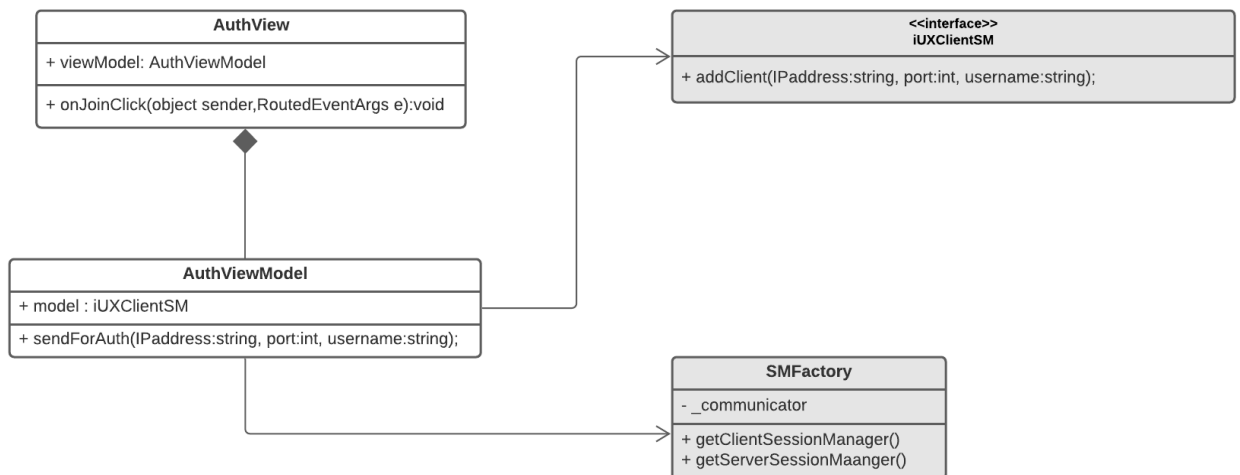
5.2 Objective

- To develop the view(using XAML) for collecting the IP address,port number and username
- To develop the view model that connects the authentication view and session management data model
- To let a user join the room if given the correct details else display an error message

5.3 Interfaces and Classes

- **AuthViewModel**: The authentication view model class.Uses an object of type *iUXClientSM* that is instantiated by the *getClientSessionManager()* method in *SMFactory* class.Using this object,the login credentials are passed to session management by making use of the *addClient(IPaddress:string,port:int,username:string)* method

5.4 UML Diagram



5.5 Activity Diagram

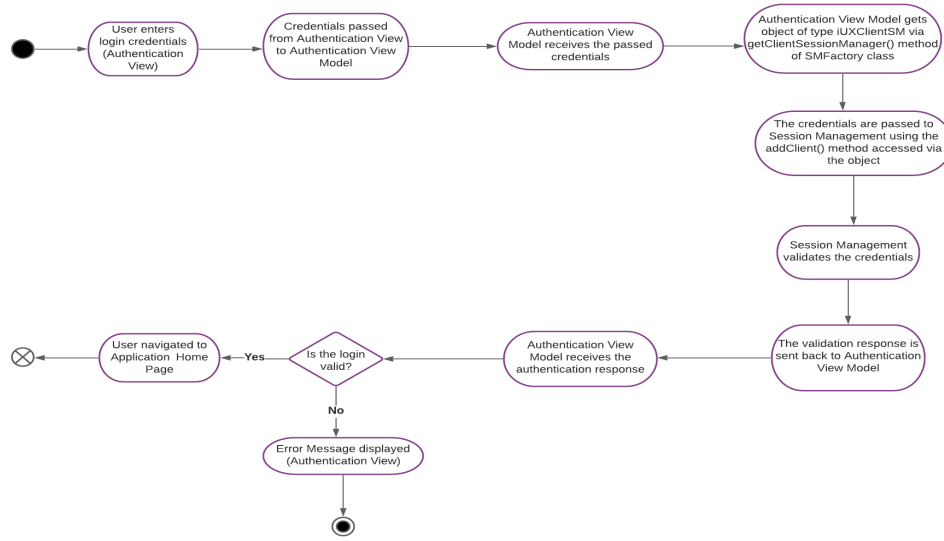


Figure 5.1: Activity Diagram of Authentication Module

5.6 Design

When a user wants to join a room, he/she should provide the credentials-IP address, port number and username-using the authentication UI provided. These credentials are passed on to the authentication view model from the authentication view. The authentication view model (class *AuthViewModel*) has an object of type *iUXClientSM* which is provided by the factory class *SMFactory*. The *addClient (IPAddress,port,username)* method is called using this object. The *addClient* method returns a boolean value denoting a valid or invalid login. If the login attempt is valid, user is taken to the home page else an error message will be displayed.

Here, the object needed to access the method *addClient* is not created by the *AuthViewModel* class as this would require authentication module to know about which class is implementing the *iUXClientSM* interface. To let the authentication module be unaware of the implementation and instantiation details, a factory class which contains methods to return objects of type *iUXClientSM* is used.

6 Home Page

Design and manage the home page, which is the first page that shows up after login.

6.1 Members

P S Harikrishnan - 111801028

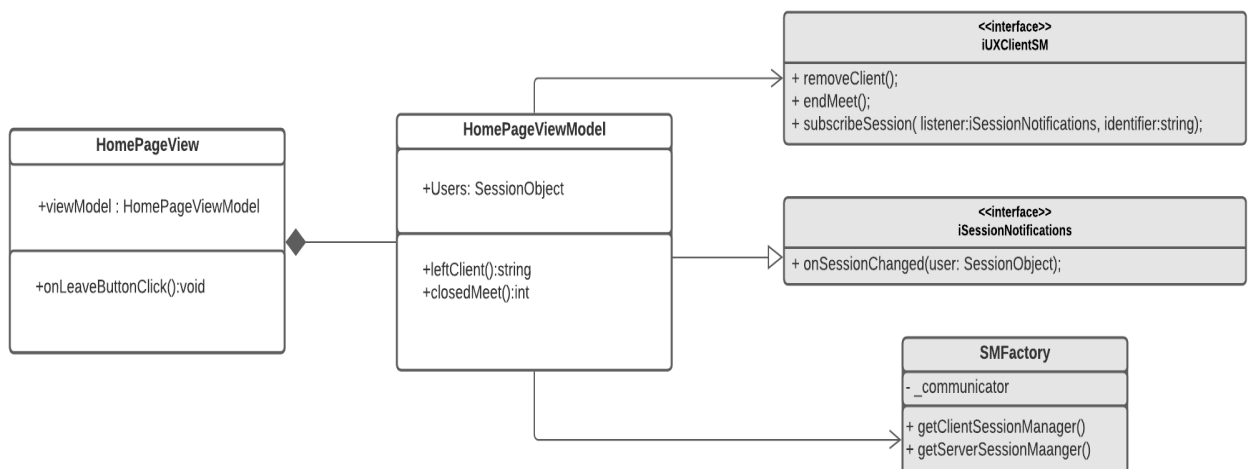
6.2 Objective

- Develop the view of the home page which shows the list of active participants along with Whiteboard or ScreenShare, and button to leave the meeting.
- Develop the View Model that connects the Home Page view and the session management data model.
- Subscribe to the session management via view model to display the list of currently active users which includes those who have left and those who have joined.

6.3 Interfaces and Classes

- **HomePageViewModel** : Class that implements iUXClientSM to send information about left users.
- **iSessionNotifications** : Interface that notifies whenever a user enters or leaves the room, so that the list remains updated.

6.4 UML Diagram



6.5 Activity Diagram

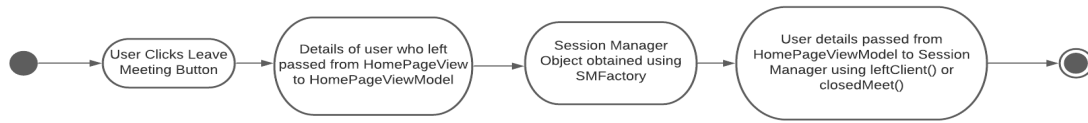


Fig1: When a user leaves the room



Fig2: When session change is notified



Fig3: New user subscribes for notifications of session changes.

6.6 Design

Once a user signs in successfully, the Whiteboard along with users list will be displayed which will act as the home page. The page will receive the users list from the Session Manager and the initial view will be created using that information. The page will also listen to notifications from the Session Manager. When a new user enters the room or when a user leaves, the new updated list of users will be obtained through *onSessionChanged*(user: SessionObject), which is a function defined in the view model. Based on this list, the view will be updated, to show currently present participants. The page will also have buttons to navigate between ScreenShare and Whiteboard pages as well to open Chat and Dashboard windows.

On clicking button to leave meeting, the user's information (userID being one of them, to identify if the user is a host or client) is passed to the Session Manager using *leftClient()* or *closedMeet()* from the view model.

7 WhiteBoard UX

7.1 Members

- Aniket Singh Rajpoot : 111801051
- Arpan Tripathi : 111801005

7.2 Objective

- Designing View of Whiteboard that mainly consists of two components: The main canvas (whiteboard) and toolbar (consisting of radio buttons and drop-downs). These toolbar would include : **Pen, Eraser, Shapes, Select, Image, Change Background, Clear Frame** options for various functions.
- Develop necessary utilities in View Model to request Whiteboard Data Model for updates on canvas from client as well as server.
- Develop necessary utilities in View Model to send updates on canvas to the Whiteboard Data Model for broadcasting to server accordingly.
- Develop necessary utilities in View Model to optimally balance between Subscribing to updates for client and sending them to Whiteboard Data Model for better real time performance.

7.3 Roles

- **Aniket Singh Rajpoot:**
 - Designing the Whiteboard front-end in a simplistic and sleek manner.
 - Handling of `onClick` events for Shape creation and deletion.
 - Implementing of locking mechanism to render fetched updates in a consistent order.
- **Arpan Tripathi:**
 - Designing the Whiteboard frontend in a simplistic and sleek manner.
 - Handling of `onClick` events for Shape modification.
 - Listening to server updates and local render requests.

7.4 Interfaces

User Interface Backend (View Model): In the view model we would define multiple classes and interfaces keeping following points in mind :

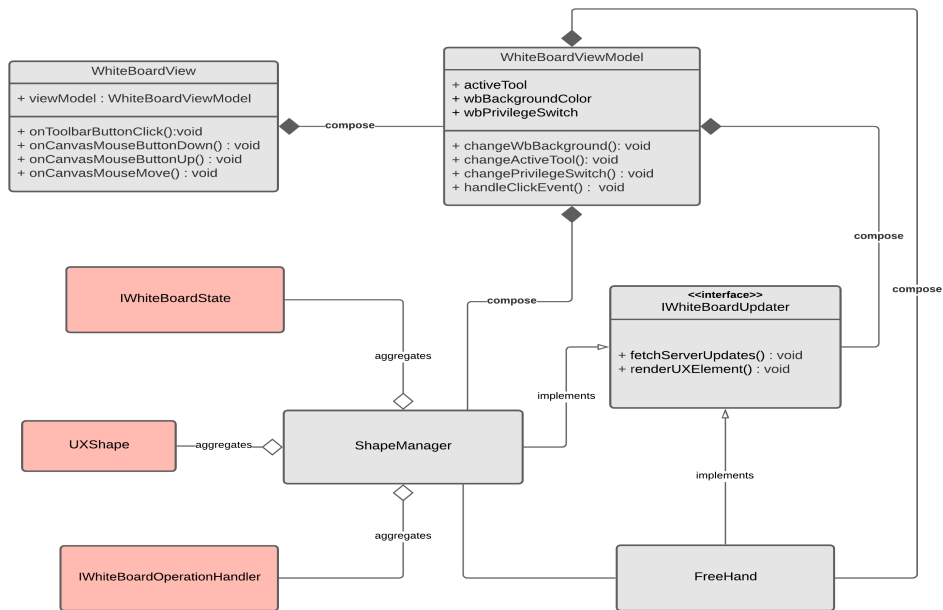
- Need to handle all the mouse events for concise calling of the methods provided by the **IShapeOperation, IClientOperation** etc. For example, we need to provide `opType` flag for the WB team to distinguish between temporary and final requests, we also need to call specific methods of **IShapeManager** for updation of shapes and specific methods of **IClientOperations** for commands like Undo and Redo.
- Need to implement a locking mechanism to render local requests by **IShapeOperation** within the `onClick` event handlers and fetched server updates by **IWhiteBoardState** with listening. In case of server latency causing an incorrect ordering of shapes, the WB module would send us appropriate updates to rectify this issue.

- **IWhiteBoardState** would provide four substantial methods for the UX, namely: *subscribe()*, *Listen()*, *saveCheckpoint()* and *getCheckpoint()*.

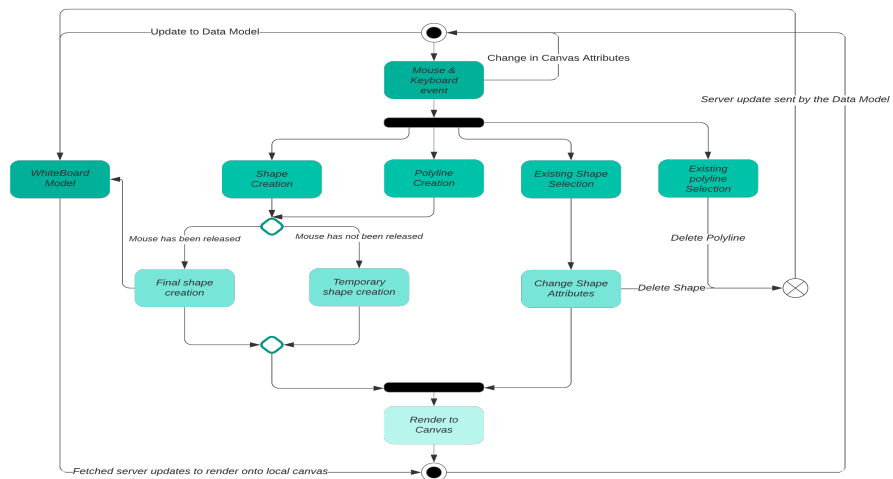
Interfaces and Classes defined in the view model :

- **IWhiteBoardUpdater** Interface: Listens to fetched server updates by **IWhiteBoardState** and local updates by **IShapeOperation** to render in ascending order of time as explained above, It also consists of the abstract methods for shape operations mentioned in the next point.
- **ShapeManager** class (would implement IWhiteBoardUpdater): This class would contain functions for:
 - **selectTool**: sets the activeTool enum maintained by us.
 - **selectShape**: sets the curShape field variable utilised by the Data Model. Multiple shape selection to be supported in Phase 2.
 - **createShape** : start, end, ShapeID, opType sent to Data Model, which utilises activeTool enum to draw appropriate shape.
 - **rotateShape** : start, end, ShapeID ,opType sent to Data Model, which utilises ShapeID variable to perform rotation.
 - **moveShape** : start, end, ShapeID, opType sent to Data Model, which utilises ShapeID variable to perform translation.
 - **deleteShape** : directly calling delete shape method provided by Data Model, which utilises curShape.ShapeID variable to perform deletion.
 - **duplicateShape** : directly calling duplicate shape method provided by Data Model, which utilises curShape.ShapeID variable to perform duplication.
 - **callUndo**: Notifies the Data Model whenever an undo command is requested by the user.
 - **customizeShape**: To adjust finer attributes like border thickness, fill color etc by the Data Model based on the selected shapes/ selected tool/ clicked button.
 - **setBackgroundColor**: Provides users the option to change background color, we would keep this color change as local, i.e, independent of all the other users.
- **FreeHand** class (would implement IWhiteBoardUpdater): This class would utilize the Shape.Polyline WPF utility to support the creation and deletion operations for freehand drawing in Phase 1. In the Phase 2, we would utilise the most suitable UIElements to provide the user with smoother free hand drawing experience as compared to the Shape class, as Canvas elements can accomodate any UIElement object as its components. This class would utilise the newer methods implemented by the WB Team who would also use the newer UIElement class other than UIElement.Shape for a better freehand drawing experience. This class would be done in Phase 2.

7.5 UML Diagram



7.6 Activity Diagram



7.7 Design

There are two UI elements namely the **Canvas** and the **Toolbar** that a user can interact with using mouse and keyboard input. These input events combined with updates from server would act as main triggers to updates on canvas. For Phase 1 we are processing shapes therefore based on selected option on toolbar the user can either create a shape or update an existing shape. We have defined a **ShapeManager** class which implements the methods of **IWhiteboardUpdater**, this class is used for performing both the functionalities of drawing new shape and modifying the existing shape using various methods provided by it. If a shape creation event is ongoing then we send update to data-model as soon as Mouse is released otherwise temporary updates are rendered directly to canvas. In an event of shape modification we update the data model as soon as shape is deleted otherwise updates are send after rendering modifications to canvas.

The freehand drawing tool would be implemented with the utilization of Shape. For phase 1 we will use the Polyline utility provided by the WPF package and implement creation and deletion operations.

Meanwhile, we also implement a locking mechanism to manage the rendering of both the local updates as well as the server fetched updates from the data model in the proper sequence of timestamps to enforce the consistency in the order of shapes across all the clients.

8 ScreenShare UX

8.1 Members

- Vinay Kumar - 111801034

8.2 Objective

- Designing a button to start and stop the screen share and to call the respective functions.
- The main goal is to render the screen sharer screen on all other client's screen who are present in the meet.

8.3 Interfaces and classes

- **IScreenShare**(Interface): It will be having one listener method **onScreenReceived(SharedScreen)**. Using this method the screenshare module is notified and starts to send screenshots in the bitmap format from time to time and render it on the client's screen.
- **ScreenShareClient**(Class): The constructor of this class will create an instance of this class, when the whole application starts running. This class has

- **isShared**: This is a data member which is boolean type and stores a True or False indicate whether the screen share started or not. This boolean value is the trigger to start and stop the screenshare part.
- **startSharing(int userid, string username)**: This method actually starts screensharing and will modify the **isShared**(set it to TRUE) and input to this method is userid and username.
- **stopSharing()**: stops screen sharing and will modify **isShared**(set it to FALSE) and displays the message”No one is sharing the screen” on all client’s screen.

8.4 UML Diagram

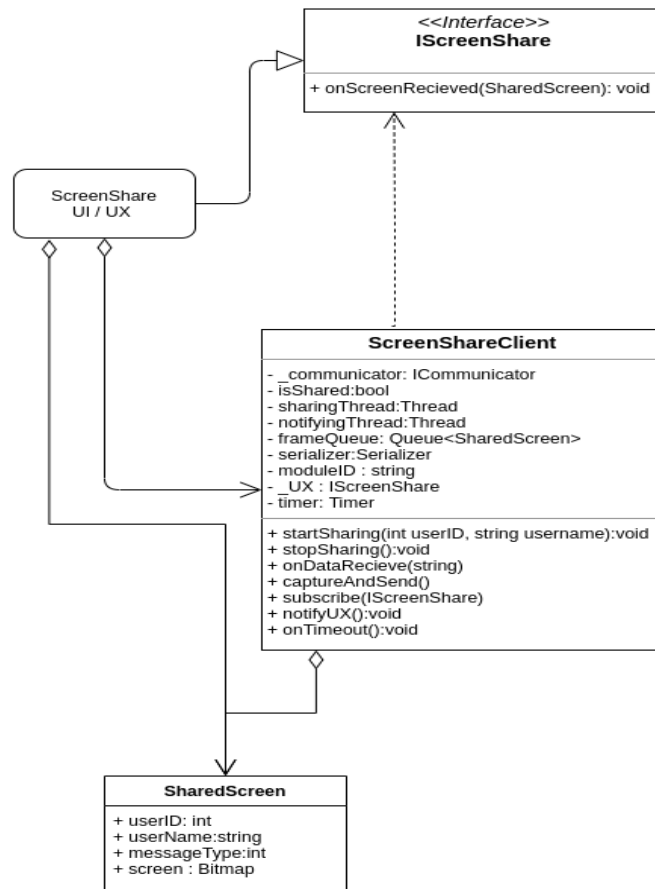


Figure 8.1: UML diagram for Screenshare module.

8.5 Activity Diagram

1. Start Sharing the Screen

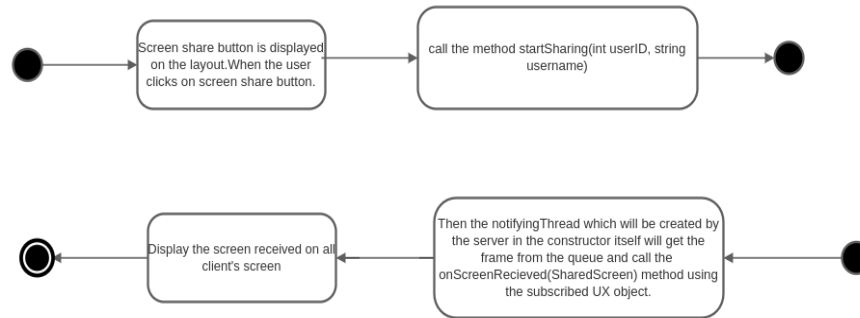


Figure 8.2: Activity diagram of Start screen sharing

2. Stop Sharing the Screen

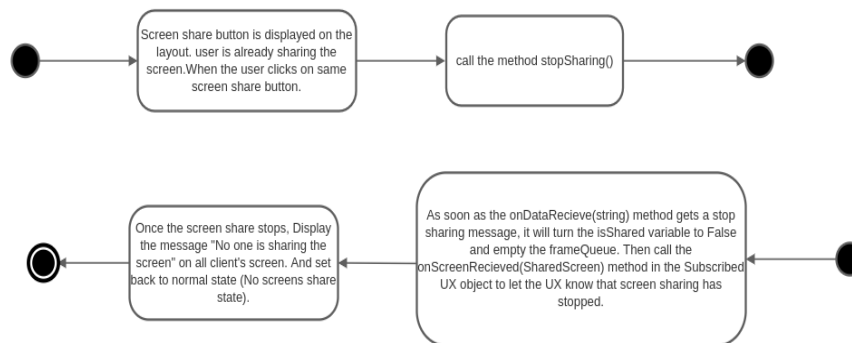


Figure 8.3: Activity diagram of Stop screen sharing

8.6 Design

- When the user clicks on the screen share button, then the method **startsharing(int userid, string username)** will be called along with the userID and username of screen sharer from the ScreenShareClient class. If the same user clicks again the screen share button, then the method **stopsharing()** will be called from the ScreenShareClient class and it stops screen sharing.
- The argument to the onScreenRecieved method is the SharedScreen object which consists of userID, username, messageType and screen shots of the screen in the

bitmap format. The messageType field is used for multiple purposes. One is if someone starts screen sharing and it is in progress, now if any user from the meet try to start screen share then by using this messageType field we can control and display a message "**You cannot share your screen**" on his screen.

- When no one shares the screen, then this message "**No one is sharing the screen**" will be displayed on each and everyone screen. If one of the member in the meet starts screen share by clicking on the screen share button, then this message "**You are sharing your screen**" will be displayed to the member who is sharing the screen at the screen share module. Rest of the members will be getting the screen sharer screen on their desktop or laptop screens.

9 Chat UX

9.1 Members

- Suchitra Yechuri - 111801043
- Sai Vipul Mohan - 111801045

9.2 Objective

- Designing a **send message box** for sending(broadcasting) user entered messages/-files to all the subscribers.
- Displaying the received messages/files along with the user's name in the chatbox and facilitating the users with a **download button** to download the files.
- Providing a **star option** in the message box, so that the users can prioritize their messages, which is included in the summary generated by the dashboard module.
- Providing a **reply option** in the message box to let the users reply to a specific message in the chat.

9.3 Roles

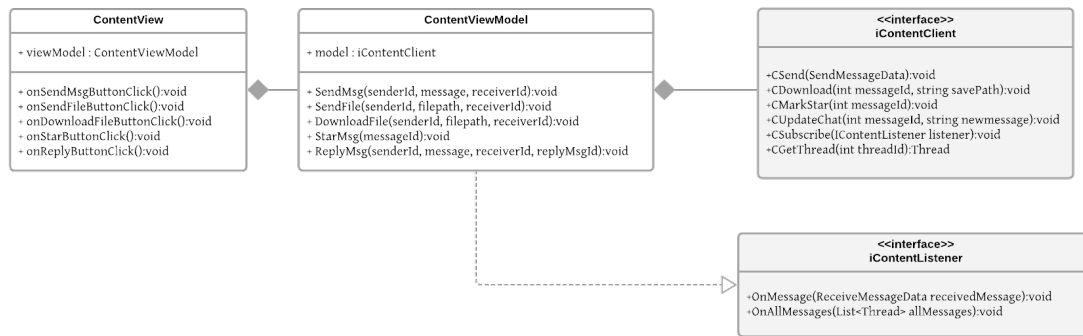
- **Suchitra Yechuri:**
 - Design the chat window in a simple and consistent manner.
 - Subscribe to content module and deal with the received messages/files.
 - Implement the reply feature provided to the users.
- **Sai Vipul Mohan:**
 - Design the chat window in a simple and consistent manner.
 - Deal with the sent messages/files.
 - Implement the star feature provided to the users.

9.4 Interfaces and classes

ContentViewModel:

- The content view model class implements the **IContentListener** interface provided by the Content module and passes this object to the **CSubscribe** function to receive messages/files from the content client.
 - The **OnMessage** function is used to receive messages/files in the form of a **ReceiveMessageData** object. This object consists of the fields for the *event type, message type, message, message id, sender id, receiver ids, reply thread id, sent time and isStarred*.
 - The **OnAllMessages** function is used to receive all the messages and files sent until now. This function is to ensure that the clients who joined new or who rejoined will be able to view all the previously sent messages. These messages and files will be received in the form of a list of thread objects, sorted by thread creation time. The **thread** object consists of the fields for *list of messages, thread id, number of messages and thread creation time*.
- To send a message or a file to the content module, the **CSend** function provided by the content module interface i.e. **IContentClient** will be used. This function takes a **SendMessageData** object as the argument. This object is constructed with fields for *message type, message, receiver ids, reply thread id*. The message type specifies whether the message is a chat message or a file. If the sent message is not a part of any existing thread then the reply thread id is filled as a special value(-1), else it is filled with the thread id of the thread to which the message belongs to.
- To download a file, the **CDownload** function provided by the **IContentClient** interface will be used with the arguments as the message id and the path to save the file in. Therefore, the content module can fetch the file from the server and save it to the given file path.
- When a message is starred by the user, the message id of the starred message is sent to the content module using the **CMarkStar** function, so that it can be included in the summary generated by the dashboard module.
- To get all the messages of a particular thread, the **CGetThread** function is used. The thread id of the thread is sent to the function in order to receive the **thread** object which consists of the *message list, thread id, number of messages and the thread creation time*.

9.5 UML Diagram



9.6 Activity Diagrams

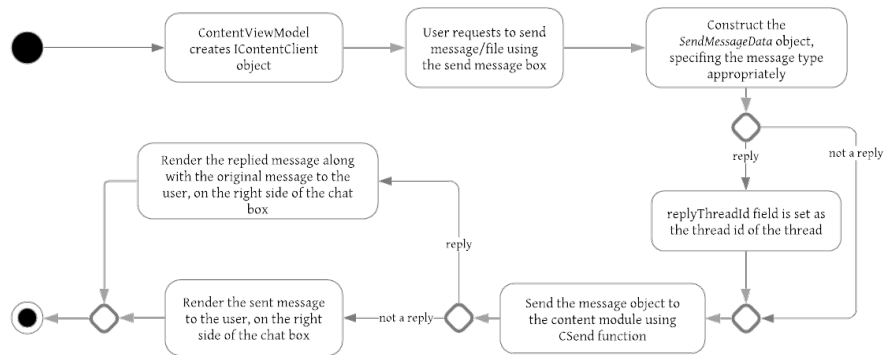


Figure 9.1: Sending messages to content team

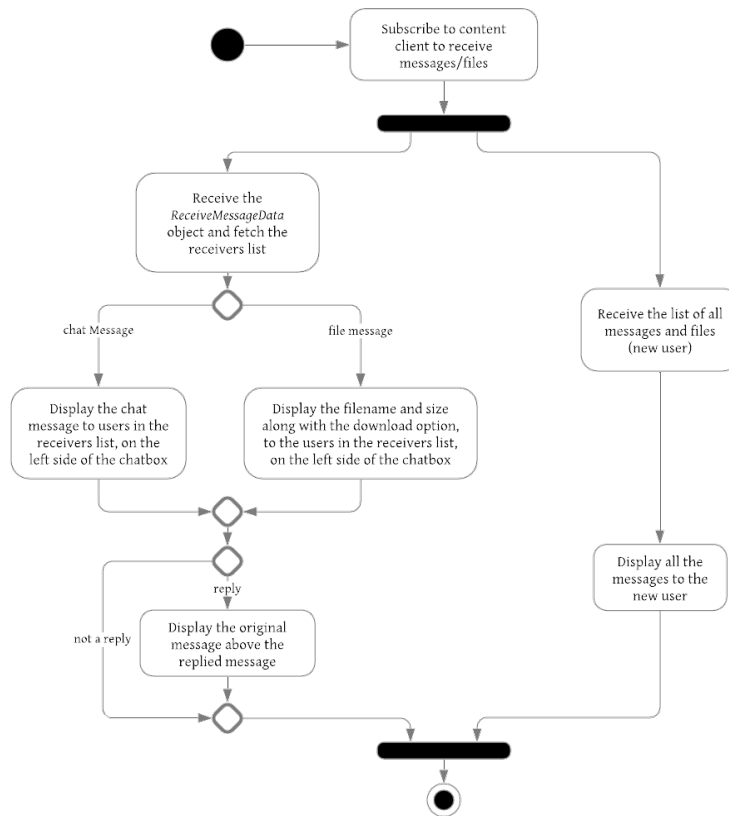


Figure 9.2: Receiving messages from content team

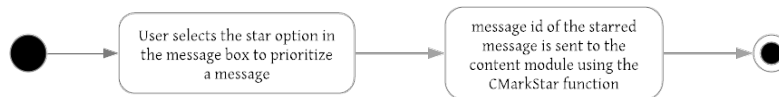


Figure 9.3: Prioritizing the messages

9.7 Design

The chat window is present on the right side of the application screen. The user can send a message with the help of the send message box which consists of the text box and the send button. The messages sent are displayed on the right side, while the messages received are displayed on the left side of the chat box along with the username. In the case of a file type message, the filename along with the size are displayed, and a download option is provided to the user. If the user chooses to download the file, then he/she is asked for the download path to save the file in. The user can also choose to prioritize a message by selecting the star option provided in the message box.

Furthermore, the user can reply to a message using the reply option in the message box, in which case the replied message along with the original message are shown to the user. The reply in thread option (*planned to be implemented at a later stage*) allows the user to reply to a message in a thread (one-level threads). The original message box consists of the number of messages in the thread, which when clicked, links to the full thread.

10 Layout & Management

Design and manage the layout of the application. Ensure that the different UI components integrate well when put together.

10.1 Members

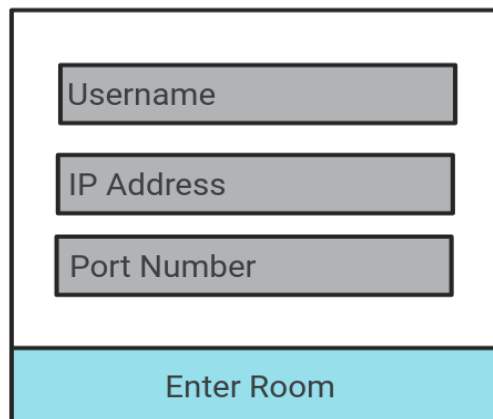
P S Harikrishnan - 111801028

Irene Casmir - 111801017

10.2 Objective

- To manage the layout of the pages and windows in the application-whiteboard,screen share and dashboard,along with participants list and chat.
- Manage the chat window when other pages are active.

10.3 Design



The diagram illustrates the authentication layout. It consists of a large rectangular container. Inside this container, there are three stacked rectangular input fields. The top field is labeled 'Username', the middle field is labeled 'IP Address', and the bottom field is labeled 'Port Number'. Below these three input fields, there is a single, wider rectangular button labeled 'Enter Room'. The button has a light blue background, while the input fields have a light gray background.

Figure 10.1: Authentication Layout

The page views are developed using XAML. Once the Authentication is done **Fig.10.1** the homepage is opened **Fig.10.2**. There will be two buttons to choose which page should be made active at any time(Whiteboard or ScreenShare) and two for chat and dashboard **Fig.10.2**. There will also be a leave meeting button away from the other

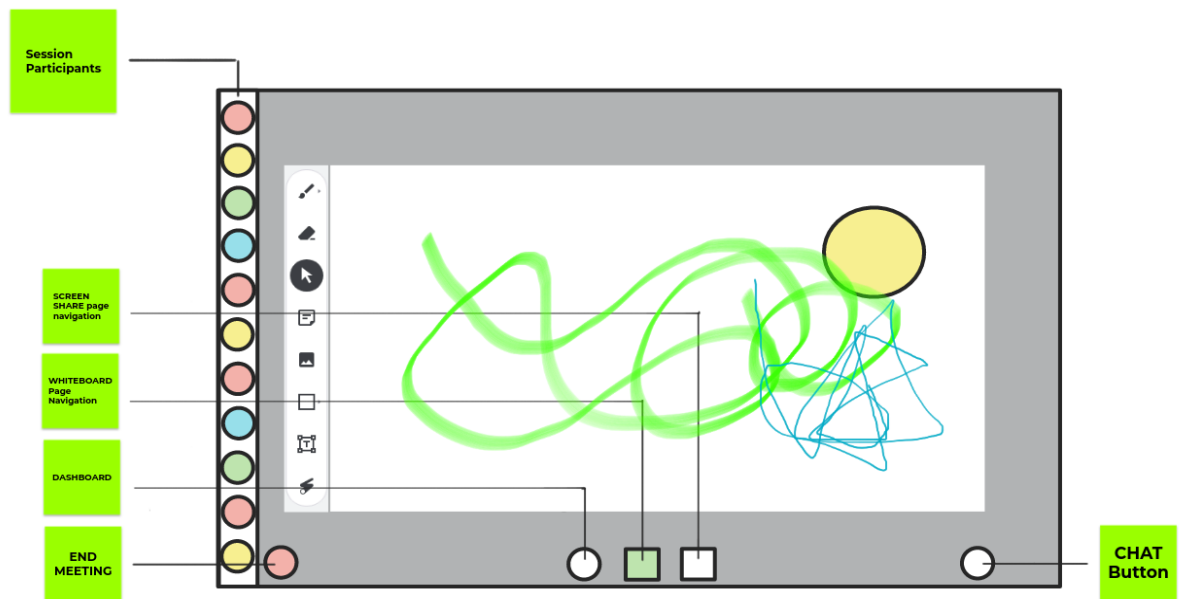


Figure 10.2: Homepage

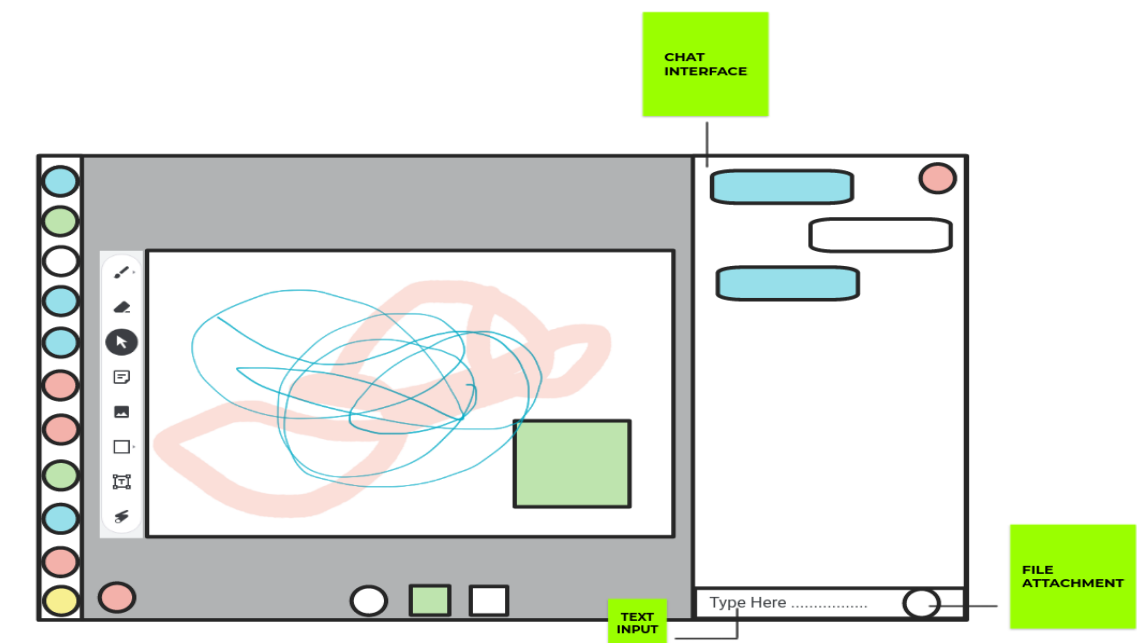


Figure 10.3: Opening Chat while on Whiteboard Page

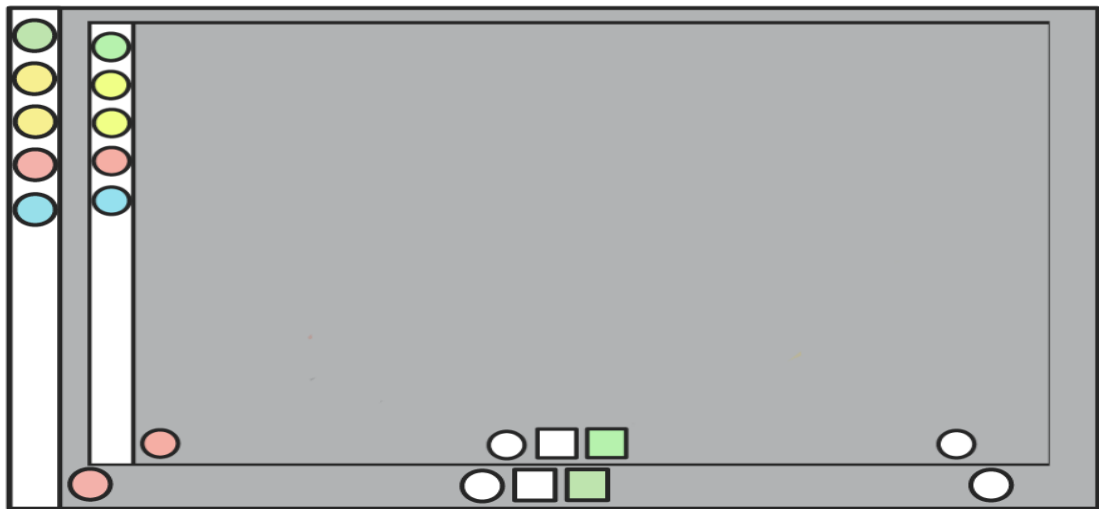


Figure 10.4: ScreenSharing

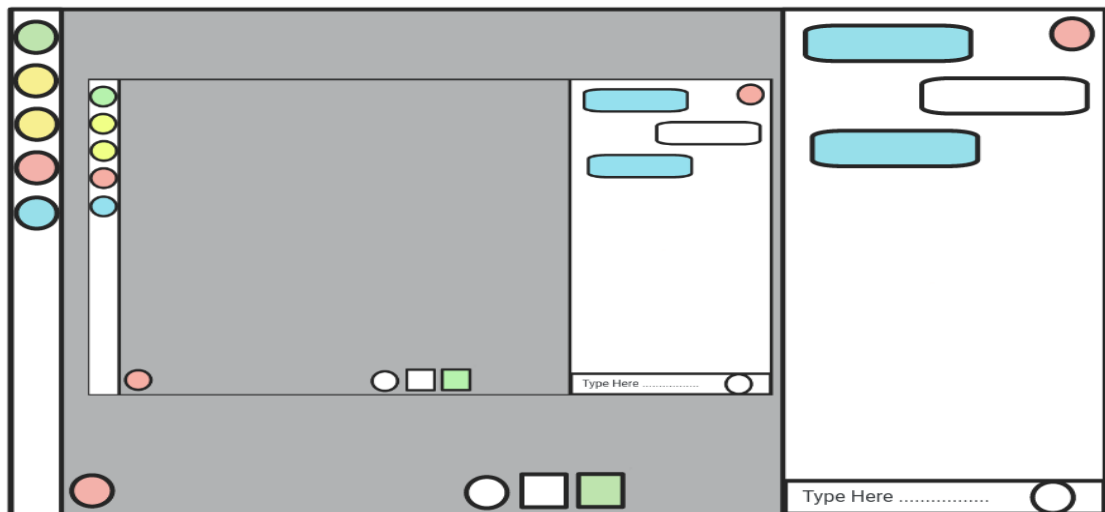


Figure 10.5: Screensharing with Chat

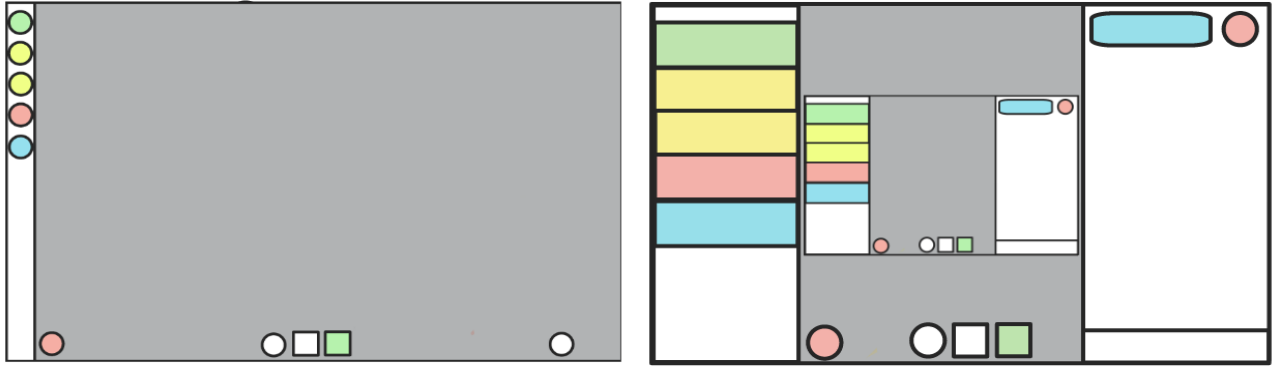


Figure 10.6: Opening Userlist and Chat on the page

buttons to reduce users from leaving the meeting unintentionally. The page navigation buttons are placed side by side for ease of navigation. Here we will use the same control for chat, leave meeting buttons as both pages share these button thus reducing redundancy. This also ensures that the same chat conversation is shown in both pages as one would like it.

The list of active users will be present as an expandable window throughout the session as shown in **Fig.10.6**. A simple click on the right column would enable users to the full users of the session participants and a scroll option will be provided in case the number of participants exceed the allotted space in that column. As seen in **Fig.10.6** The space occupied by chat, main page and the user list has to be re-rendered by giving highest space occupancy to the current page contents (WhiteBoard or ScreenShare) followed by chat interface and then the userlist interface.

The chat window comes at the right side of the current page **Fig.10.3** and once again its size will depend on which page is active at that time.

The application screen is made compatible with all screen resolutions by setting the width and height of the application window to appropriate bindings. The dimensions of the containers are not set to fixed values but to relative sizes. The dashboard is a separate window so that the summary of a meeting can be viewed anytime irrespective of which page is active in the application window.

It must be noted that the images in this section are used to give an idea of the overall layout over the different components and the pictures by themselves do not depict the exact appearance of the application it self. They are subject to change based on implementation efficiency and user-experience.

11 DashBoard UX

11.1 Members

- Mitul Kataria - 111801025

11.2 Objective

- Design UX for dashboard module which provides the insights about the meeting.
- The insights include the detailed analysis of users' activity such as graph of Number of users vs Time, User vs Messages sent by that user in the discussion, total number of participants present in the meeting at that point of time, etc. and the summarized text of the discussion happened so far in the session which is obtained using summary logic.
- The key objective is to build up an aesthetic UI using above mentioned information that would be accessible by any attendee at any point of time throughout the meeting.

11.3 Interfaces

```
// DashboardViewModel Interface for DashboardView
// Processes the data obtained from session manager
// Includes methods for fetching summary and
// breaking down the UXTelemetry object into required format
public interface DashboardViewModel
{
    // returns total number of participants joined in the current session
    int GetTotalParticipants();

    // returns the latest summary generated by the summary logic sub module
    string GetSummary();

    // returns the byte array plot of Number of Users vs Time
    // which is generated using processed data points provided by UXTelemetry object
    byte[] GetUsersvsTimePlot();

    // returns the byte array plot of User vs Number of messages sent by that user
    // which is generated using processed data points provided by UXTelemetry object
    byte[] GetUsersvsMessagesPlot();
}
```

Figure 11.1: DashboardViewModel

```

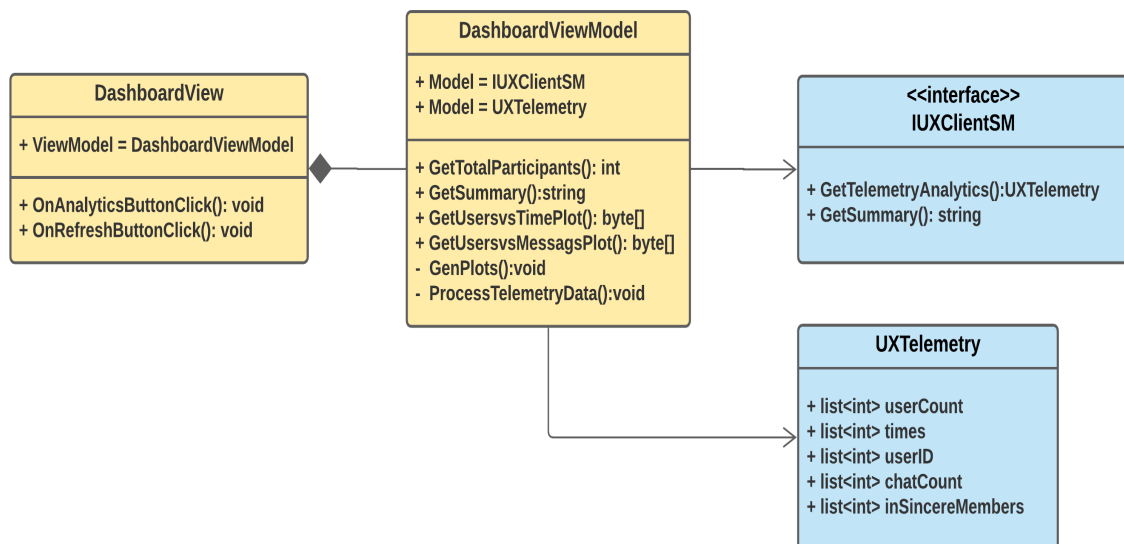
// DashboardView interface that contains event listeners which gets triggered on button clicks
// On each trigger the dashboard data gets rendered into the WPF application
public interface DashboardView
{
    // event listener on analytics button
    void OnAnalyticsButtonClick();

    // event listener on refresh button
    void OnRefreshButtonClick();
}

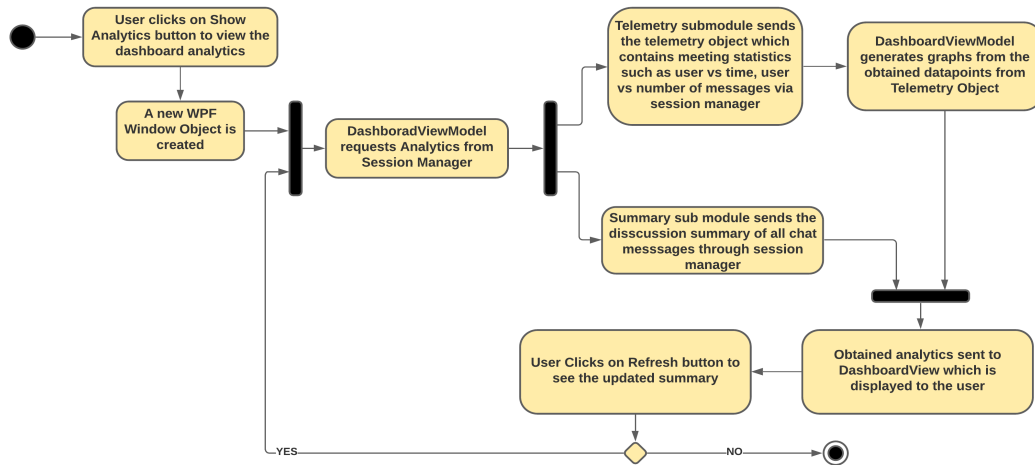
```

Figure 11.2: DashboardViewModel

11.4 UML Diagram



11.5 Activity Diagram



11.6 Design

- The **Show Analytics** button would be present beside the leave meeting button at the bottom part of the main application window (as can be seen in the layout and management section) using which the users can get the insights about the meeting.
- When a user clicks on the Show Analytics button, the event listener method **On-AnalyticsButtonClick()** will get triggered and a new window will pop up which will display the analytics.
- The obtained data that includes the key insights about the meeting which is generated by summary and telemetry submodules would be fetched by the **DashboardViewModel** and then sent to the **DashboardView** to display it to the users.
- Text summary would be generated by the summary logic sub-module using some Natural Language Processing techniques on all chat messages of the session. This will be obtained using the **GetSummary()** method of the **IUXClient()** interface.
- The telemetry sub module would provide an object which will contain the various insight data. This data will be obtained using the **GetTelemetryAnalytics()** method which returns the object of **UXTelemetry()** class.
- The methods **GetTotalParticipants()**, **GetUsersvsTimePlot()**, **GetUsersvsMessagePlot()** break down the data obtained from the **UXTelemetry** object and thus make it easier for **DashboardView** to handle them.

- **GetUsersvsMessagePlot()** method generates the graph from the obtained data points of userID and charCount attributes of the UXTelemetry object and then renders it into an image of byte array which will be used for plotting the graph.
- **GetUsersvsTimePlot()** method generates the graph from the processed data points number of users vs time intervals and then renders it into an image of byte array which will be used for plotting the graph in the DashboardView.
- **GetSummary()** method returns the summary text using the GetSummary() method provided by the IUXClientSM interface.
- The data of the object will be then processed further before displaying. The **ProcessTelemetryData()** method will process the data points and convert them into a format that is suitable for generating the graphs. Then, the **GenPlots()** method of **DashboardViewModel** will generate the graphs from the processed data points.
- In the new popped up window, there will be a Refresh button using which the users can see the latest insights about the session. When a user clicks on this button, the event listener method **OnRefreshButtonClick()** will be triggered which will update the content of the same dashboard window and it will not pop up another window.
- As the WPF Window object itself comes with Close button functionality, the Close button is not explicitly designed in the UX.
- Using **subscriber-publisher** design pattern for listening to the change in summary or telemetry data would be inefficient as updating data on each session change would be redundant in case when no user is clicking on the Show Analytics button. Therefore we decided to go with simple method triggers on clients' requests.
- Thus, the dashboard data would be obtained only when a user requests it, that's when he clicks on the Analytics button or Refresh button.
- One bottleneck of this design might be the time it takes to generate the summary when the chat messages count grows rapidly. In order to deal with such time delay issues, one solution might be to use the loaders in the DashboardView until the summary is obtained or another solution might be to display the previously obtained summary text or any other better solution we could come up with during the code implementation.

12 Module Level Interaction

