

---

# Meet.me

## **Module Specifications: WhiteBoard and ScreenShare Module**

Manas Sharma

Team Lead(WhiteBoard and Screen Sharing)

---

## Contents

<b>1. Introduction</b>	<b>4</b>
<b>2. Objectives</b>	<b>4</b>
<b>3. Team</b>	<b>5</b>
<b>4. Individual Roles</b>	<b>5</b>
<b>5. Workflow and Intra-module Dependency</b>	<b>7</b>
<b>6. UML Diagram</b>	<b>10</b>
<b>7. Activity and Flow Diagram</b>	<b>12</b>
<b>7. Whiteboard Operations submodule (Owner: Parul Sangwan)</b>	<b>13</b>
7.1 Objectives	13
7.2 Design	13
7.3 Features provided	15
7.4 Design Choices:	15
7.4.1 Internal Storage of Objects	15
7.4.2 The output format provided to the UX module	17
7.4.3 Toggle between Active and Inactive States:	18
7.5 Input and Output Specifications for Real-time rendering	19
7.6 Real-Time shape creation/modification logic	20
7.7 Operations	21
7.8 Additional Design Patterns Used	22
7.9 Challenges	22
<b>8. State Manager submodule (Owner: Ashish Kumar Gupta)</b>	<b>23</b>
8.1 Overview	23
8.2 Objectives	23
8.3 Design	24
<b>9. Checkpoint Handler submodule (Owner: Chandan Srivastava)</b>	<b>32</b>
9.1 Objectives	32
9.2 Design	32
9.3 Activity diagram:	34
9.4 UserLevelHandler Class:	35
9.5 Extended goal:	35
<b>10. Board Server Communicator submodule (Owner: Gurunadh)</b>	<b>36</b>
10.1 Overview:	36
10.2 Objectives:	36

10.3 Design:	36
10.4 Class Diagram and Activity Diagram	37
<b>11. Screenshare submodule (Owner: Neeraj Patil)</b>	<b>43</b>
11.1 Overview:	43
11.2 Objectives:	43
11.3 UML diagram:	44
11.4 Design Analysis:	44
11.5 Activity Diagram:	47
<b>12. Interfaces</b>	<b>48</b>
<b>13. Future Scope</b>	<b>52</b>
<b>14. Features Delivered</b>	<b>53</b>

## 1. Introduction

---

During this post-pandemic era, the ability to collaborate remotely has become one of the most essential requirements in any industry. At such a moment, utility tools like Whiteboarding and screencasting applications come to the rescue. In this project, we are working on the module to implement some basic functionalities for each of the two utilities.

## 2. Objectives

---

We currently plan to implement the following features for the first phase of the project:

1. Provide a Whiteboard module interface(to be used by UX module) to support the following draw features:
  - Draw standard shapes like lines, rectangles, ellipses, polylines(free-hand curves).
  - Select shapes on the Whiteboard and perform operations like(limited to standard shapes):
    - Translation
    - Rotation
    - Resizing
    - Colour-change operations, stroke width, shape fill
  - Deletion of shape objects from the Whiteboard
2. Provide undo-redo capabilities on the client side to restore any changes.
3. Provide the client, the functionality to toggle the Whiteboard state from active(edit-mode) to inactive(read-only mode)
4. Provide username-tags for a drawn shape when the user hovers over a shape.
5. Provide the ability to checkpoint the current state of Whiteboard and be able to restore the Whiteboard to an already checkpointed state.

### 3. Team

---

**Manas Sharma** - Team Lead

**Parul Sangwan** - WhiteBoard Submodule

**Ashish Kumar Gupta** - WhiteBoard Submodule

**Chandan Srivastava** - WhiteBoard Submodule

**Gurunadh Pachappagari** - Network Interface for both WhiteBoard and Screen Share Submodule

**Neeraj Patil** - Screen Share Submodule

### 4. Individual Roles

---

#### → **Manas Sharma (Team Lead)**

##### **Roles:**

- To collaborate with team members to decide upon the abstract design for the module.
- To handle integration of the module with other modules and resolve conflicts.
- Check performance of the module and do unit testing.

#### → **Parul Sangwan (Board Operations)**

##### **Roles:**

- Handle the interface for creating shape objects and operations on the created shape objects and provide it to the UX module.
- Implement the functionality to freeze/unfreeze the current Whiteboard access to the current client.
- Implement classes which would represent the shapes that are drawn on the whiteboard.

#### → **Ashish Kumar Gupta (State Manager submodule)**

**Roles:**

- Implement an interface to start Whiteboard state manager and subscribe to changes needed to be provided to the UX team and simultaneously listen to client-side whiteboard communicator to get updates and notify the UX about the updates.
- Perform state management operations in an easy and efficient manner on both client and server side.
- Provide the logic undo-redo operations.

**→ Chandan Srivastava (Checkpoint and User Access Level submodule)****Roles:**

- Implement the interface to store the current state of Whiteboard(on server) in the form of checkpoints.
- Provide the ability to restore the Whiteboard state to a previously saved checkpoint.
- Implement a *UserLevelHandler* interface to set the permissions for a client to modify shape objects created by other users.

**→ Gurunadh Pachappagari (Board state communicator submodule)****Roles:**

- Implement a communicator submodule to interact with the state manager of the whiteboard and the network module.
- Responsible for serialization of *BoardServerShape* objects to be sent through the network from the client to the server. Also handle the deserialization of the same at the client's end after receiving through the server and passing to the state manager.
- Also responsible for serialization and deserialization of Bitmaps received from the Screenshare submodule.

**→ Neeraj Patil (Screen share submodule)****Roles:**

- Responsible for creating the interface to capture the screenshots at intervals and transmitting them to the server for further transmission to the session clients.
- Find a way to optimize the process by reducing redundancy in capturing the

screenshots for parts of screen where changes are insignificant

## 5. Workflow and Intra-module Dependency

---

In this section, the overall workflow is explained through the workflow diagrams.

The following diagram shows the flow when changes are received from the UX end of the client and how the changes are reflected all across the module.

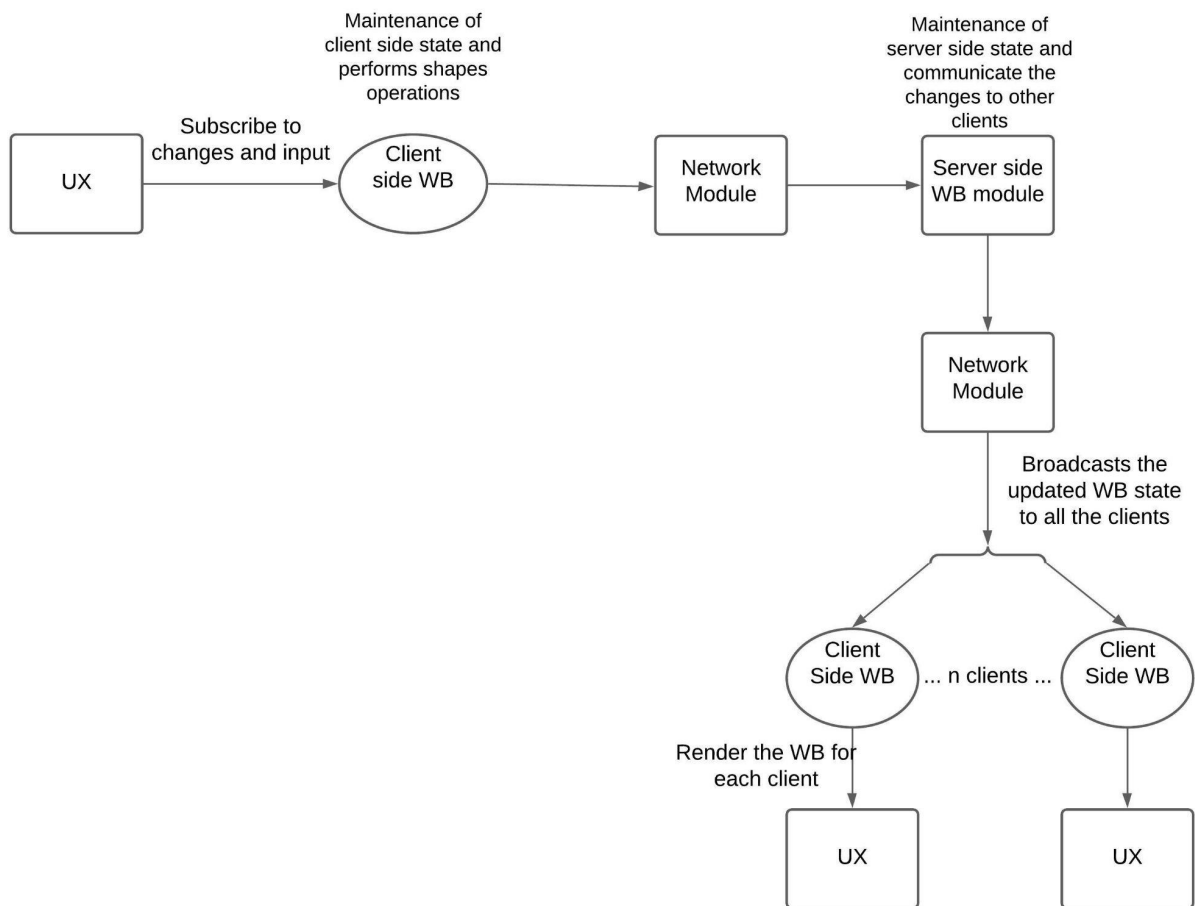


Figure 1 : Workflow for WhiteBoard submodule

The following diagram shows the flow when input is received from the UX end of the client to start sharing the screen and how the changes are reflected.

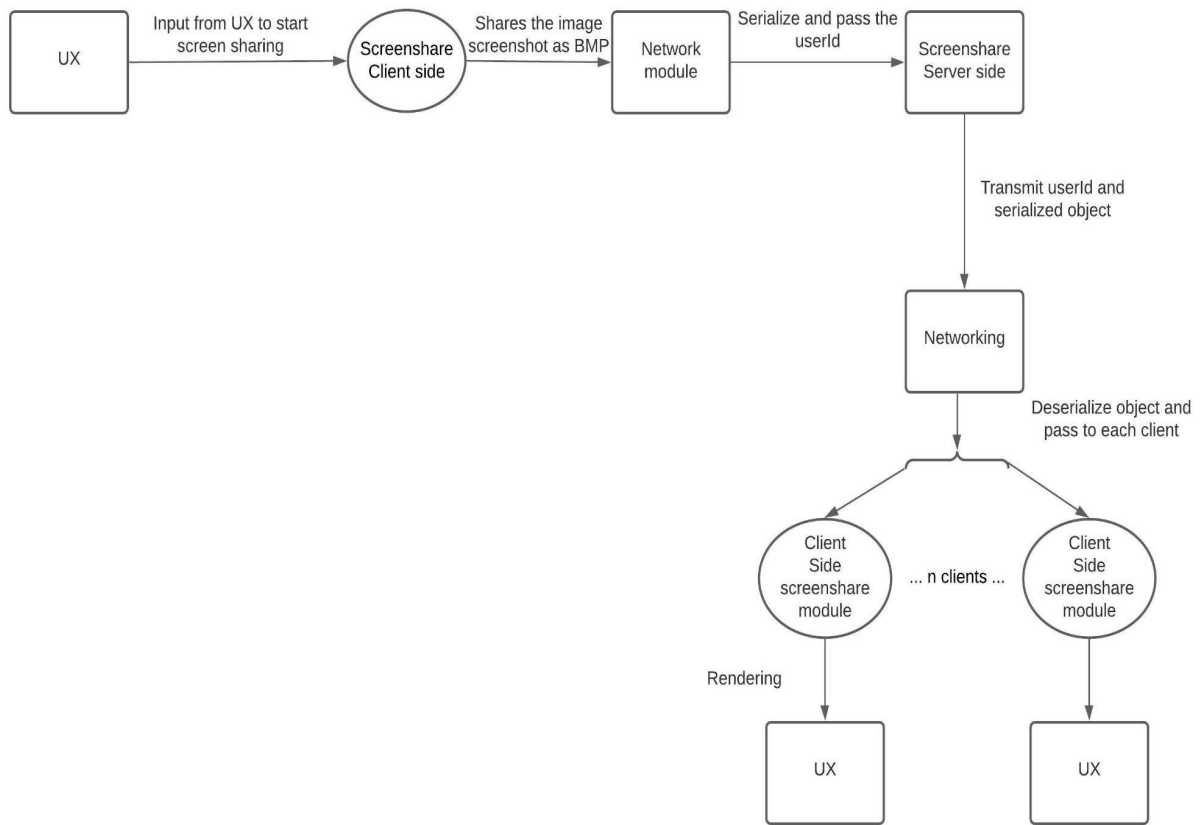


Figure 2 : Workflow for Screenshare submodule



The following intra-module dependency diagram depicts how the submodules are structured and shows the inter dependencies within the submodules on both the client and the server side.

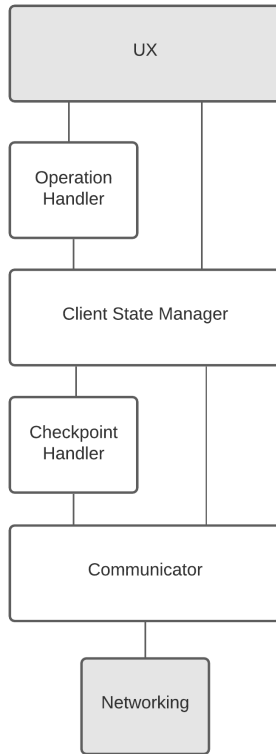


Figure 3. Intra-module dependency(client)

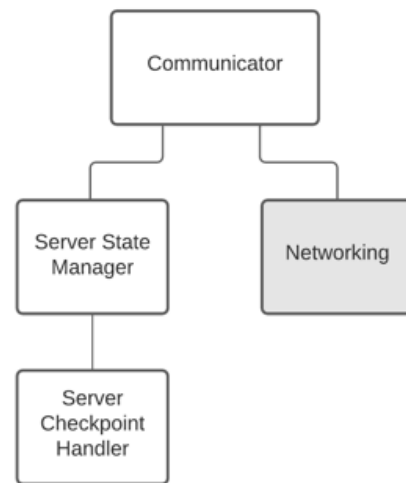


Figure 4. Intra-module dependency(server)

## 6. UML Diagram

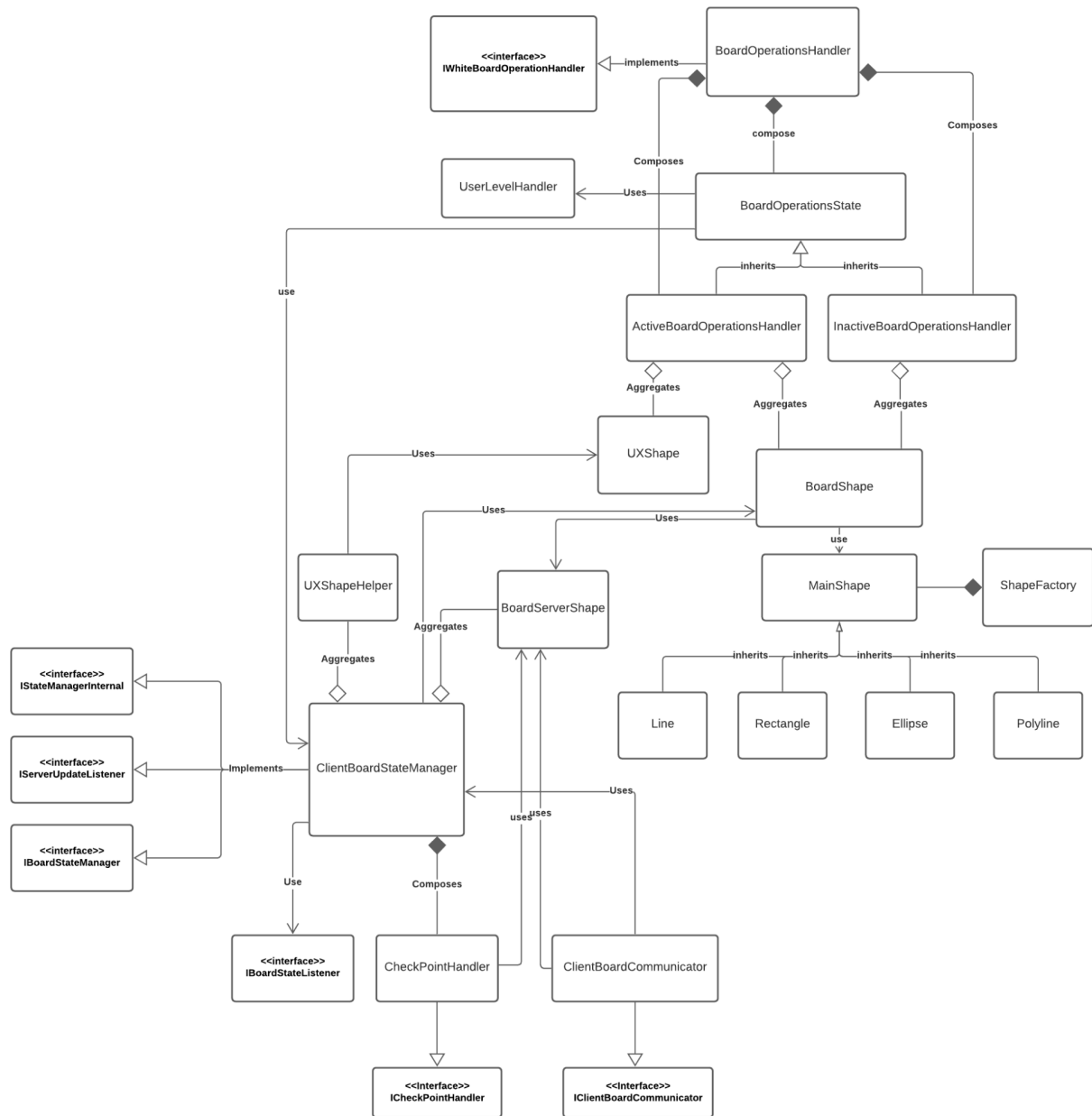


Figure 5: UML diagram for client side operations for Whiteboard module

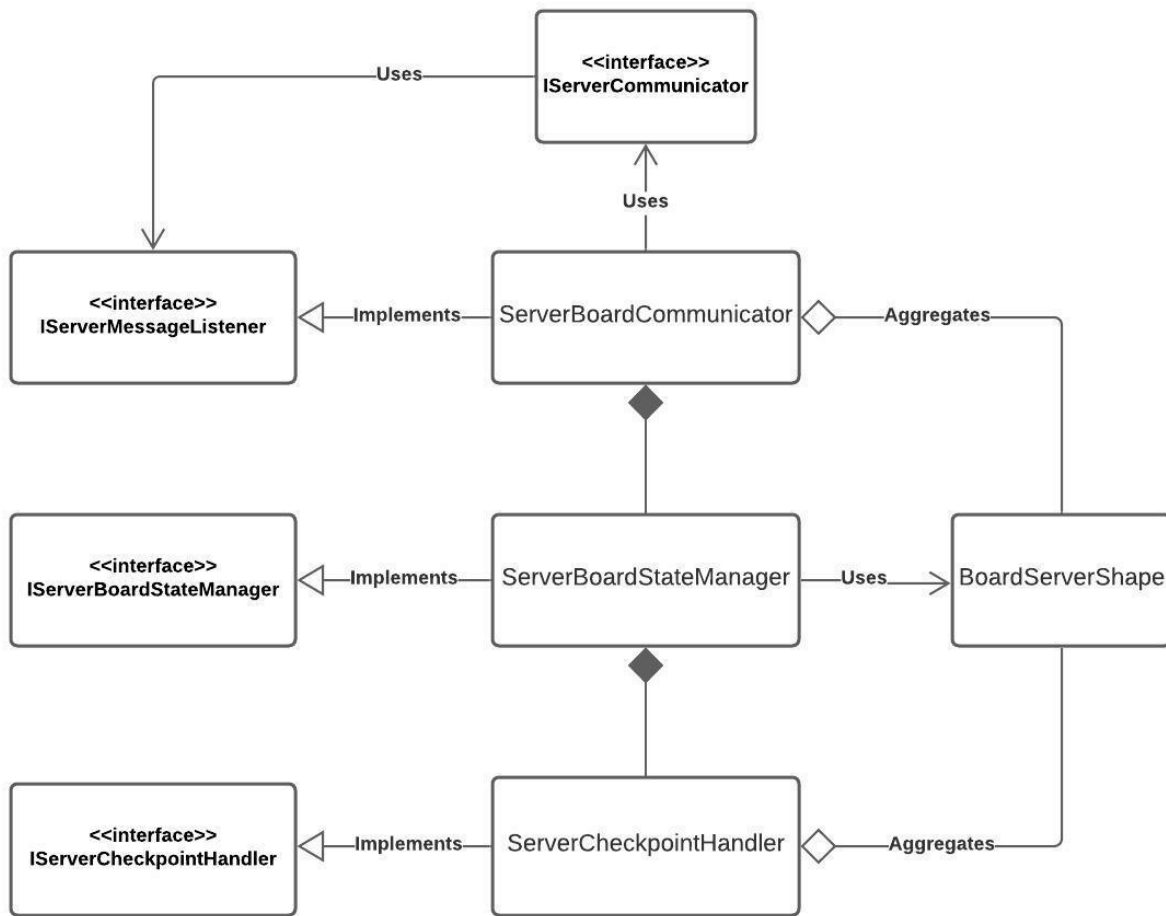


Figure 6: UML diagram for server side operations for Whiteboard module

## 7. Activity and Flow Diagram

In this section, we have broken down the activity flow for various cases for the Whiteboard submodule. Due to the sheer size of the work being done in each submodule, the following section just discusses the overall activity flow in Whiteboard. The detailed analysis is done with the specific submodule.

As for the Screenshare submodule, the activity flow is drawn within the submodule specs only.

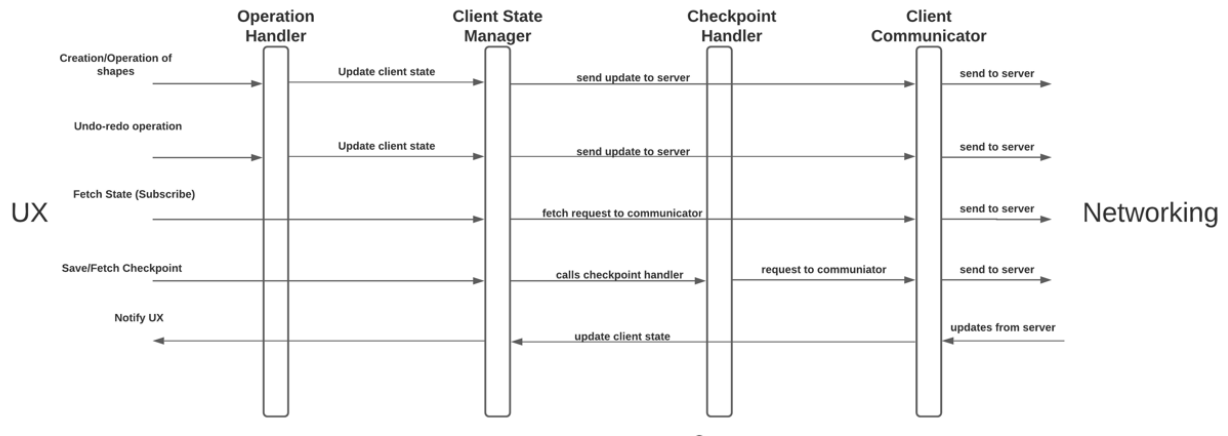


Figure 7: Activity flow diagram for Whiteboard(client side)

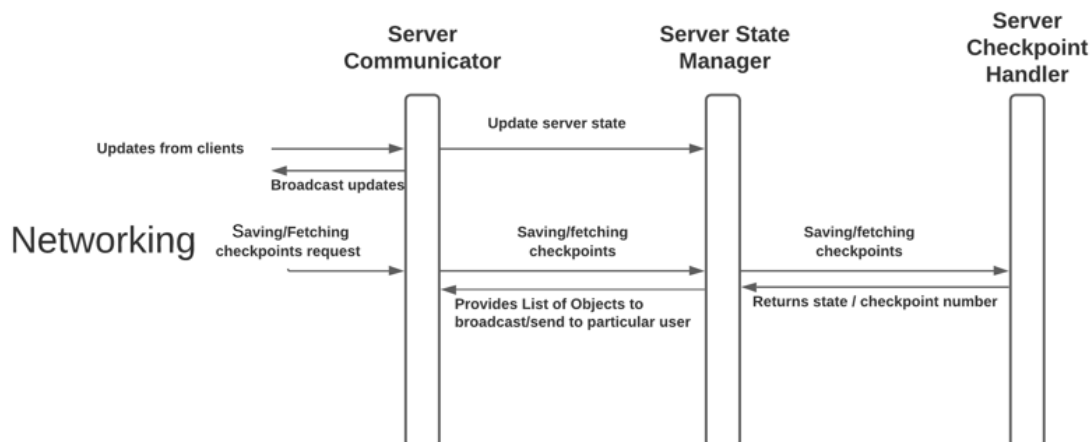


Figure 8 :Activity diagram for Whiteboard(Server side)

## 7. Whiteboard Operations submodule (Owner: Parul Sangwan)

---

### 7.1 Objectives

#### Shape Operations and Creation

- Provide an interface to the UX module for creation, deletion, undo, redo, or performing shape modifications on the UX for instance resizing, translating, rotating or other modifications like shape fill, freezing/unfreezing the whiteboard, etc, along with additional methods to get user ids' from shapes.
- Implement this interface for performing operations on stored Board Object objects along with updating the state stored in the Client-Side State Manager and eventually sending the updates to the server.
- Providing UX the feature to real-time render the creation or modification of shapes. This involves maintaining internal state while an operation is being performed.

#### Internal storage of shapes

- Implement classes which would represent the shapes that are drawn on the whiteboard.
- Maintain and implement the Classes for the objects that would be used for communication from Whiteboard to UX and from WhiteBoard Client to the Server.
- Writing the logic for operations on the Shapes.

#### Freezing and Unfreezing Functionality

- Implement the freeze (blocking Operations on a particular Client) and unfreeze functionality on the WhiteBoard.

### 7.2 Design

#### 1. Overall design

The UX module would be provided an interface that would create the objects and perform operations on that object. Additionally, the whiteboard of the user can be in either of the two states, chosen by the user himself:

1. **Inactive** - The client won't be able to make any changes himself, but will be able to see the changes of other people on his whiteboard.
2. **Active** - The client can actively participate in making edits to the shapes or creating shapes on the whiteboard.

Based on whether the board is in Active or Inactive state the main Handler class exposed to the US will automatically call appropriate methods to update the object and correspondingly update the Client-side State Manager.

The whole state and identity of the shape present at the whiteboard will be represented by an object called BoardShape. These shapes will be sent to the server as another object of the class named BoardServerShape that contains a list of BoardShape's. This is to accommodate the case when there are multiple updates in a specified order that need to be sent to the server.

## 2. UML Diagram

**Note:** The grey-colored classes belong to other team members. Apart from this there are classes like **BoardColor**, **Coordinate** etc would also be present but are not shown in the diagram to avoid making the Diagram congested.

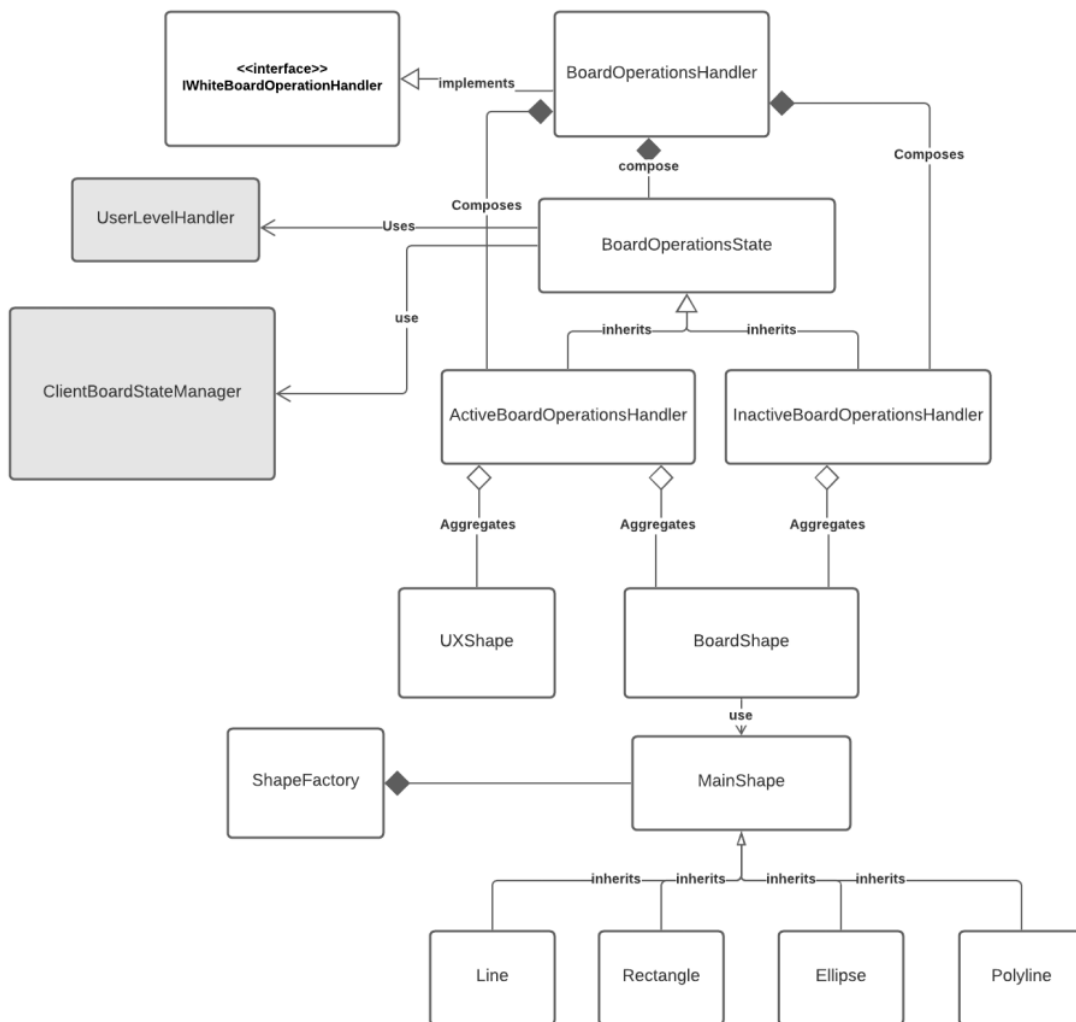


Fig 1: UML Diagram for Shape Operations

## 7.3 Features provided

- a. Supported shapes and their representations which are stored in a whiteboard module.
  - i. Ellipse - Height, width, center
  - ii. Rectangle - Height, width, center
  - iii. Polyline - A set of points, representative of freehand drawing.
  - iv. Line - center, angle of rotation and width
- b. Supported operations on shapes
  - i. Translation
  - ii. Rotation
  - iii. Resizing
  - iv. Changing Height/Width
  - v. Changing color, stroke width, shape fill color
  - vi. Deletion of shape
- c. Other operations:
  - i. Undo/Redo (handled by state Manager)
  - ii. Finding UserId corresponding to shape provided
  - iii. Switching state between Active and Inactive states.
  - iv. Privilege switches based on user level.

## 7.4 Design Choices:

### **7.4.1 Internal Storage of Objects**

- The attributes that define a shape are stored in the object of the class **MainShape**. This class contains the height, width, center, starting position, angle of rotation etc and is enough to completely define a shape. There might be some properties that are useful for some shapes but are not useful for other shapes, for instance polyline requires the class to store the points on the drawn curve, and hence this attribute will be initialized to **NULL** itself in case the polyline is being created.

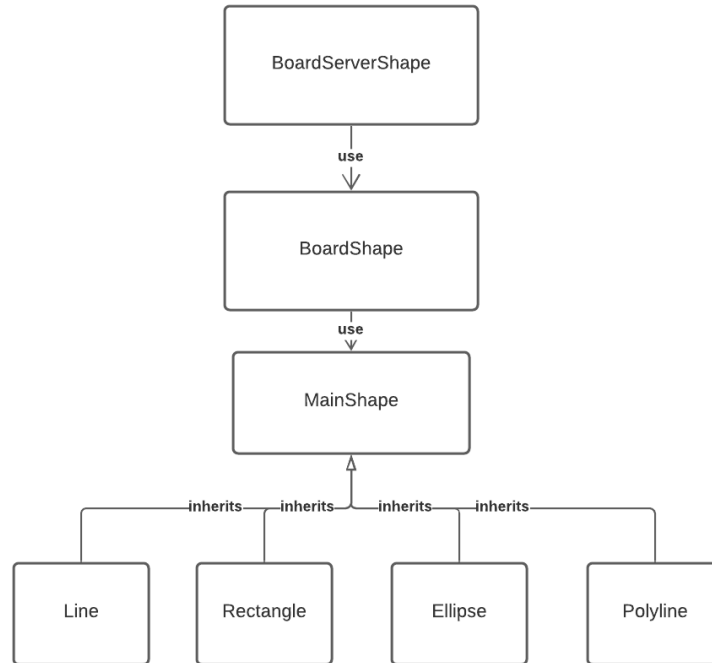


Fig 2: Hierarchy for objects used to represent shapes

- All the shape classes **Line**, **Polyline**, **Ellipse**, **Rectangle** inherit from the class **MainShape** which is defined to be abstract so that it cannot be used as it is without any shape identity associated with it. This class contains the code for functionality that is common across most shapes. The specific child might override the functionality of the provided virtual functions in case it wants to be specific or different with the implementation of the methods.
- This object defines the identity of the shape, however there are other attributes that are required to be associated with the shape that has been rendered on the UX for instance the user-id of the creator, the time of creation/modification etc. Hence, this **MainShape** object is not directly stored at the state Manager. Another class **BoardShape** is stored at the server, which encloses all this information. This information is important to store since they are used for processing shape at server and client side state-manager.
- The object sent to the server is **BoardServerShape**. Objects of this class contain a list of **BoardShapes** that need to be sent to the server for processing.

**Reason :** The decision for having this class instead of directly sending the **BoardShape** to the server is because there are multiple operations whose updates consist of multiple **BoardShape** objects. In case these operations are sent as single objects they might reach the server in an unpredictable order, thus distorting the order in operations required for performing an operation itself.



- Systems.Windows.Shapes class is used both at the UX and the Whiteboard. The Whiteboard module provides the UX with pre-built objects of the class Systems.Windows.Shapes enclosed inside another object of the class named **UXShape**. This class along with the shape consists of operations that UX needs to do with the shape. For instance, a DELETE flag indicates that the Systems.Windows.Shapes provided is already present on the UX and needs to be deleted. Similar, is done for the CREATE flag. This modular way of storing and managing, helps to associate other properties with a shape in later stages of the project.
- A deep copy of the received **BoardShape** object from the state Manager is first created in the shape Operation Handler before any modification to the shape is done by handler.  
**Reason:** This is done so that the state of the Manager and the objects stored in the Managers are never inadvertently destructed.
- **UXShape** Class contains constructors to obtain UXShape objects directly from the object stored in the form of **Mainshape**.  
**Reason:** This was possible because there were only a limited number of features of a shape that needed in order to be correctly rendered on the canvas. Hence whenever an update to the UX, would just require calling the constructor of UXShape with the overloaded constructor taking input the **MainShape**.

#### 7.4.2 The output format provided to the UX module

- From the perspective of the UX team, only 2 operations exist: DELETE, CREATE. These are the only two flags that the operation handler will associate with the shape being sent to the UX. All the operations are modelled in the form of DELETE and CREATE operations from UX point of view.
- The output to the UX is a list of **UXShape** objects. These objects have their associated flags: CREATE/DELETE, along with the amount the shape should be rotated and translated.
- The coordinate system for the Whiteboard module is set as the left-bottom point on the canvas. Also the module treats the WhiteBoard canvas as an infinite plane.

**Reason:** Systems.Windows.Shapes cannot directly store the location of the shape on the canvas, and the angle at which it is rotated. Hence these are separately sent as a field inside the **UXShape** object.

**Creation - CREATE flag:**

For the Creation operation, UX receives a CREATE flag.They should render the Systems.Windows.Shapes as it is on the Canvas, and perform translation and angle as specified in the object sent to them.

**Deletion - DELETE flag:**

The UX extracts the id of the object from the received **UXShape** object. They then find that object in the canvas and remove it from there. The whiteboard module does not directly send the object present at UX by reference.

**Reason :** This was done, so that there is no pointer sharing between UX and Whiteboard module, to avoid any inadvertent changes to the states, in case UX mistakenly modifies the shape present on their canvas.

**7.4.3 Toggle between Active and Inactive States:**

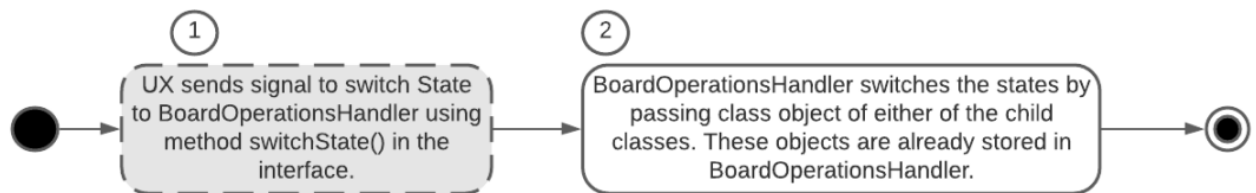


Fig 3: Activity Diagram for switching whiteboard states (Active/ Inactive)

**Design Pattern Used** - State Pattern. This pattern allowed the removal of multiple if-else statements in the **BoardOperationsHandler**, to check whether the whiteBoard is in an Active state or inactive state.

In the UML Diagram, the **BoardOperationsState** is a Base Class whose child classes **ActiveBoardOperationsHandler** and **InactiveBoardOperationsHandler** have methods about how to deal with operations in this state. On the **SwitchState()** method, the **currentState** object of class **BoardOperationsState** is toggled and the state is changed.

## 7.5 Input and Output Specifications for Real-time rendering

BoardOperationsHandler receives operations on shapes or states related to the whiteboard. The variable shapeComp is taken as a parameter from the UX indicating whether the UX has observed a mouse-up event or not. This is because until the mouse-up event, the state is not finalized and the updates are not sent to the server. The operation is performed on the object, and a series of operations are then sent to the UX module in the form of **List<UXShape>** to render on the whiteboard. The state handler maintains local memory (temporary **BoardShape** object stored in the class **ActiveBoardOperationsHandler**) about the last operation that was performed. When the shapeComp function parameter is received by the handler from the UX, the local buffer is cleared, the state manager is notified of the update and the handler becomes ready for the next operation.

There are 2 types of operations that require real time processing, they are

### **1. Creation Operation:**

At the start of Creation UX sends a shape Id of null, and gets the updates. In these updates it also receives the ID that is given to the shape locally by the operation handler. Now, in order to use the real-time functionality offered, they need to send this shapeld that they obtained on the next call of the function. Real time continues till shapeComp = true indicating that the rendering is complete, and the operations handler can now clear its local buffer and send updates to the state manager.

### **2. Modification operation:**

The process for real time rendering is the same as the creation operations. However, in this case, the handler would ask the state manager to give the shape that is stored corresponding to the id of the shape that has been asked to modify.

- Other operations like delete, shapeFill color change are not needed to be real-time rendered.

## 7.6 Real-Time shape creation/modification logic

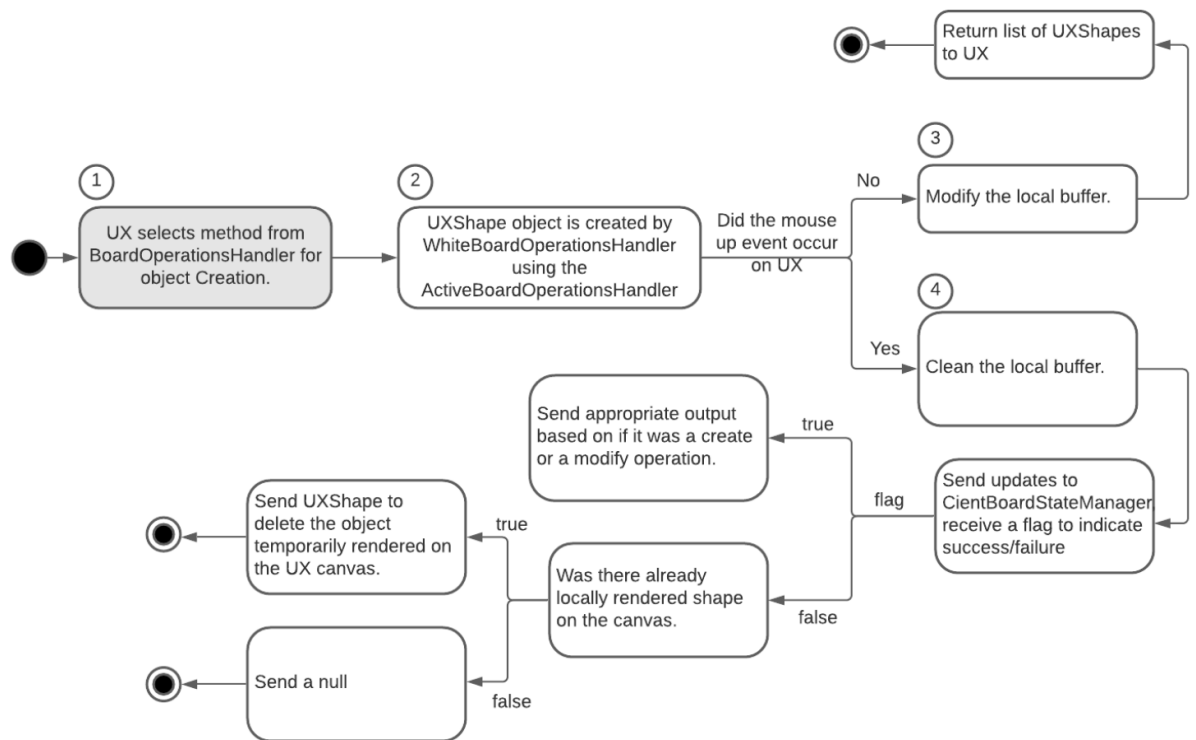


Fig 4: Activity diagram for real time creation/shape operation

- The above Activity diagram assumes the state is Active.
- When UX sends input to the BoardOperationsHandler(1), it would have the currState object as either an object of ActiveBoardOperationsHandler or InactiveBoardOperationsHandler. It would call currState.\_method\_() where the \_method\_() depends on the operation it is(2).
  - For creation, the shape-specific attributes are calculated on the basis of the start-end coordinates provided as method parameters, and a BoardShape object is created in case it is just the beginning of the real-time operation that the client on the UX is performing. This object then is stored locally in the handler, for serving the next bit of the ongoing real-time rendering. After the first call, the locally-stored object, say \_lastDrawn, is fetched and modified as per the new call, and appropriate CREATE/DELETE **UXShape** objects are sent to the client on UX. When shapecomp is true, then these updates are sent to the server.
  - In case of modification the object with the provided shapID is fetched from the stateManager and stored locally in the handler. Any further request for render of on-going operation is then served from this locally

stored update. On **shapecomp** = true, the update is sent to the state Manager who in turn processes this request. For the Undo-Redo operation, the ClientBoardStateManager itself performs Stack Updates, and returns the list<**UXShape**> to the Handler, which is directly sent to the UX.

- If the client-side state manager returns a true, then only the updates are sent to the UX, else appropriate action is taken depending on whether there is some rendering that already took place on the UX, because of the real-time rendering functionality. This object is then deleted, and new is rendered if needed.

## 7.7 Operations

### **Translation:**

Just shifting the center of the state stored. Polyline represents a free hand curve, and so it cannot be moved.

### **Rotation:**

The angle of rotation from the x axis (--->) is stored. The angle is always kept between -180 and 180 degrees.

### **Resizing:**

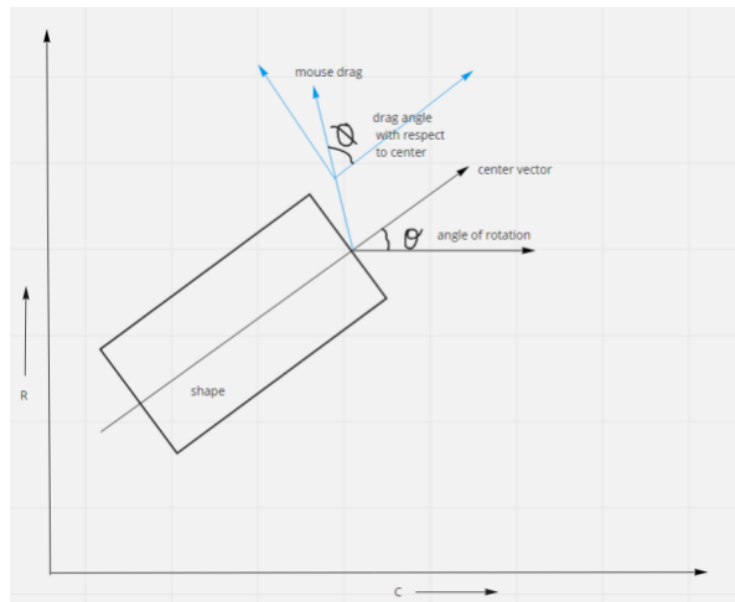


Fig 5: Resize operation on Rectangle.

Based on the angle of rotation of the shape  $\theta$  from the c axis, the displacement components along the rotated axis at  $\theta$  are calculated and depending on which latch the resizing is happening, addition and subtraction to the height and width of the original shape are done.

There is a slight modification in the way the line is resized. For a line, on a resize operation the width is increased, that is, if the above figure the rectangle is considered as a line (squeeze the height), then the  $\cos \Phi$  component is discarded.

## **7.8 Additional Design Patterns Used**

### **1. Factory pattern.**

There is a central factory, which on shape creation requests, and an identifier, creates the required object (of Line/Ellipse/Polyline/Rectangle class) and returns that object of **MainShape** class to be used for saving state at manager.

**2. Prototype pattern :** The **MainShape** class contains a clone() method with all the shape classes (Line, Ellipse, Rectangle and Polyline) inherit and override.

**3. State Pattern** - mentioned above.

## **7.9 Challenges**

1. The System.Windows.Shapes was not directly serialisable and hence we could not directly store the output in this form.
2. Some params needed to be given to UX separately in the object UXShape, since this information could not be enclosed inside the System.Windows.Shapes object.

## 8. State Manager submodule (Owner: Ashish Kumar Gupta)

---

### 8.1 Overview

State management is one of the key requirements in any Whiteboarding tool which allows multiple users to collaborate at real-time. The aim of state management is not only to manage state at the backend of the client to save current updates and support operations like undo/redo, but also to keep the whiteboard states of all clients in sync. This design specification document provides a glimpse of how this whiteboard state management is planned to be done.

### 8.2 Objectives

- **Client-side**
  - **Provide Interfaces:** Interfaces to start a whiteboard state manager and subscribe for changes need to be provided to the UX whiteboard team. An internal Interface needs to be provided to the whiteboard shape operations handler as well.
  - **State maintenance:** Maintain data structures such that they hold current state and perform state-updates efficiently.
  - **Notify UX:** Listen to client-side whiteboard communicator to get updates and notify UX about the updates.
  - **Undo-Redo:** Build logic for undo-redo operations, such that a client can only undo-redo its own changes.
- **Server-side**
  - **Provide Interface:** Interface for server-side state manager class needs to be provided to server-side whiteboard communicator to maintain a common state across all clients and if required create a checkpoint. Note that, saving the state in the form of a checkpoint will not be done by the state manager, it will be done by the checkpoint handler in our team.
  - **State maintenance:** Maintain data structures such that they hold current common state and perform state-updates efficiently.

## 8.3 Design

### Client Side:

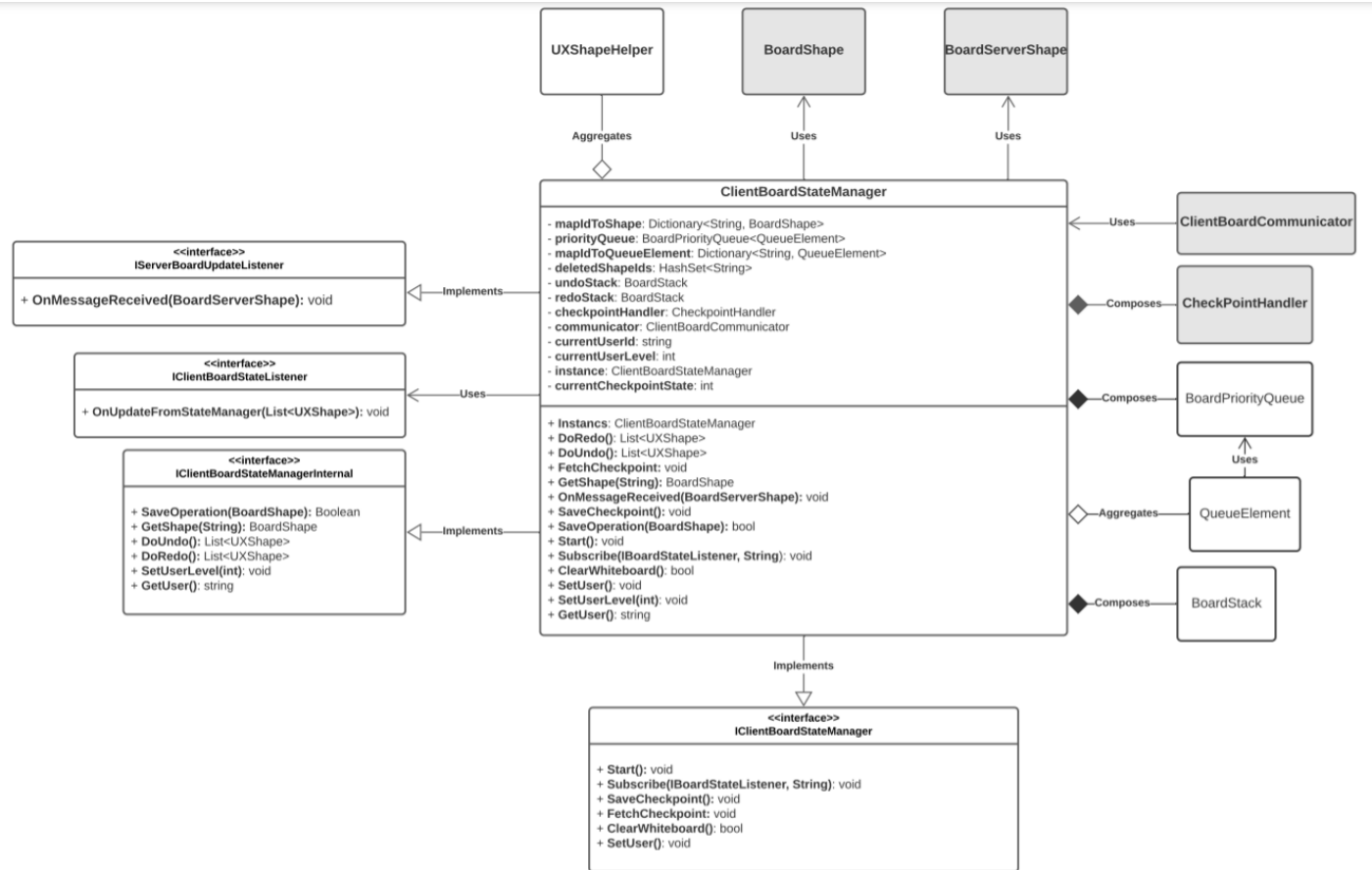


Figure 1. Client-side class diagram for Whiteboard State Manager: Including interfaces and state managing data structures

### How is the state maintained?

At client-side, the whiteboard state management will be done using **ClientBoardStateManager** class. To maintain the current state of Whiteboard, it uses four data structures to store **BoardShape** objects. These **BoardShape** objects contain the complete details of a shape and its last modification time, which is there on the whiteboard. These four data structures are –

1. **A Map from BoardShape ID to BoardShape:** This **BoardShape** ID (String) which acts as a key to map, will be the same as that of Shape's (UIElement) ID. This form of storage is needed because each **BoardShape** object will have its own unique ID, and the operation handler when needed to do some operation will require this **BoardShape**. To get the **BoardShape** in constant time, it's better to have a Map like this, otherwise in a List form of storage it will take linear time.



2. **A Priority Queue of QueueElement (BoardShapeID, Timestamp, Index):** There will be a Priority queue (max heap) in which each node will contain three elements, BoardShape ID (String), timestamp (DateTime) and index (index in heap array). The priority will be based on timestamp. This is needed to maintain the whiteboard state in sync with the server's whiteboard state. The reason for that is, let's say there are two clients A and B are whiteboarding together. A drew a circle at 10.0 seconds and B drew a square at 10.1 seconds at the same place in the whiteboard. Now the update from the server, of A's circle, reaches B at 10.2 seconds (200ms latency considering networking overhead). If we do not maintain an order with respect to timestamp, A's circle will be above B's square, which will not be a case at server's state. So, to handle this, UX must delete B's square temporarily, then add A's circle and then again add B's square. To do this we need to know all the shapes which have a timestamp greater than A's circle. We can't use a linked list because getting to that element will take linear time. Instead, if we assume there are  $k$  such elements then Priority Queue takes  $O(k \log n)$  and in general since these kinds of very close updates will be less, so  $k$  will usually be 1.
3. **A map from BoardShape ID to QueueElement:** Since each operation on a shape makes it most recent, whenever an update for a particular BoardShape comes, its timestamp also needs to be updated and hence the Priority Queue needs to be reordered. But to search for the node to update in Priority Queue, search on basis of BoardShape ID will take linear time and then the update and reordering will take another  $O(\log n)$  time. But if we can directly access the node and update it, the time will be logarithmic only and that can be done using this map.
4. **HashSet of IDs of deleted BoardShapes:** In order to maintain synchronization between client thread and server update thread this extra storage is required. Consider a case when a client starts operation on a shape it created. He just started working on it at UX-Real time rendering and this update is yet to come to StateManager. However another client has deleted the same shape and his update has reached and acquired this client's state lock. He deletes and this update is yet to go to the UX and at the same time the current client who was working with the same shape at real time releases his mouse button. Now the request to modify will come and will raise an exception but it's not an exception since the server update has not reached UX yet and the shape is visible. So to handle this case I have kept an additional data structure to know which IDs belong to deleted shapes. Once recreated (on undo - redo), these IDs are removed from the HashSet.

*Note: As can be seen from above operations, for all the shape operations, create, delete, modify, the state update will take logarithmic time. In terms of space-complexity, the first data-structure which is a necessity takes  $O(kn)$  where  $k$  is the size of a BoardShape object, while other two – a priority queue of (String, DateTime) and map from string to pointer are around space overhead provides us a better time complexity and to simulate real-time online applications, this trade-off is quite manageable.*

**Undo-Redo Stack:** These stacks will only be at client side since a client can undo or redo only his/her modifications. Each element of these stacks will be a pair of BoardShape objects. The first object in the pair will be an object before modification and the second object will be the object after modification. In create operation, the first object in the pair will be NULL while in delete operation the second object will be NULL.

Since BoardShape objects are themselves stored in stack, a limit on stack size is necessary otherwise physical memory of the device will take a serious hit. So, to maintain 7 latest changes in stack (since undoing up to 7 times is supported), implementation of these stacks will be done using double ended queues, which will automatically drop the 8<sup>th</sup> last change upon insertion of the latest change.

When undo is called the element popped is first pushed to redo stack. Then UX is directed to delete the second BoardShape object and insert the first one in its place. This restores the previous state. However, it's not this much trivial, because all the updates which have happened from server end and have a higher timestamp will be first undone and then this deletion and insertion will occur and then again, the undone changes are redone.

So, undo-redo operation for a relatively quiet client, with active other members, will take a  $O(n \log n)$  amount of time. Here  $n$  is the no. of shapes in current state.

*Note: All these data structures will be static, so they don't change from one instance to another.*

---

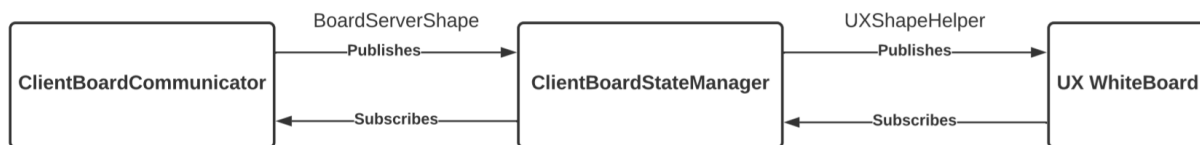
## **Relationship with other components**

The BoardOperationHandler uses ClientBoardStateManager to undo/redo, get a BoardShape object and update current state on Shape operation/creation.

Whiteboard UX, will subscribe to this ClientBoardStateManager in order to receive any notification of updates from server end, which will come through this class. So, this manager class will act as publisher for Whiteboard UX. So, between ClientBoardStateManager and Whiteboard UX, there is a publisher-subscriber relationship. UX needs output in the form of

UXShape, hence ClientBoardStateManager will send them updates in the form of a list of UXShape objects. The requests for fetching/saving checkpoint and clearing whiteboard from UX will also be entertained by ClientBoardStateManager only.

ClientBoardStateManager will subscribe to ClientBoardCommunicator to get notifications of some server update. So, ClientBoardStateManager will act as a subscriber. Hence, here again there is a publisher-subscriber relationship however this time the class is acting like a subscriber. The ClientBoardCommunicator provides objects in the form of BoardServerShape, so ClientBoardStateManager composes an instance of Convertor class to convert this to BoardShape.



**Figure 2. Relationship between UX, State Manager and Communicator:** The communication objects are stated as well

### Why this design choice?

The ClientBoardStateManager follows a Singleton pattern as the state should be the same throughout.

We are following a two-way publisher subscriber pattern, in which the ClientBoardStateManager is the publisher for UX and a subscriber for ClientBoardCommunicator. For undo-redo and state maintenance, we are using enhanced data structures. We are using this two-way publisher subscriber pattern because we don't want the state manager to communicate with the networking module and worry about serialization and deserialization. Similarly, since UX needs to be notified of the server updates, it has to subscribe to this class to receive notifications for those changes. This not only provides a proper work distribution and modularization but also is better in terms of time complexities as well as space overheads involved. For example, we are not using the Memento pattern, which is well known for its state maintenance and rollback, because according to its design, it leads to saving of state at each change. That means we have to store an array of states which will be a huge space overhead. Even if we don't consider our complete state as memento-state, and just consider the BoardShape objects as memento-state then also our purpose of keeping undo-redo only for this client's operations is defeated since then we won't be able to figure out whether the last update in the state was from this client or a server update. We won't use Observer-Observable pattern, because although their functionalities are mostly same, we want to have asynchronistic design and doesn't want the publisher-subscriber to know each other, so that UX doesn't need to address the notification at the same time and it happens in a message-queue manner.

## Activities at Client-Side (Involving only WhiteboardStateManager)

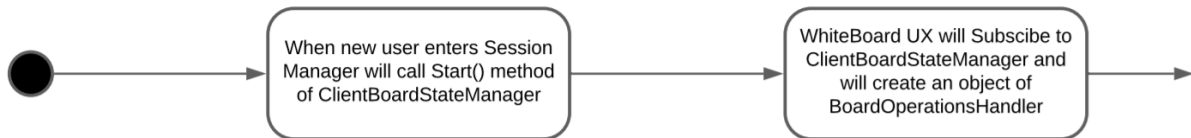


Figure 3. Activity when App Starts

Session Manager on the start of the App will call Start() method of ClientBoardStateManager which will lead to initialization of all the state data structures and will make the ClientBoardCommunicator to subscribe to the Networking module.

---

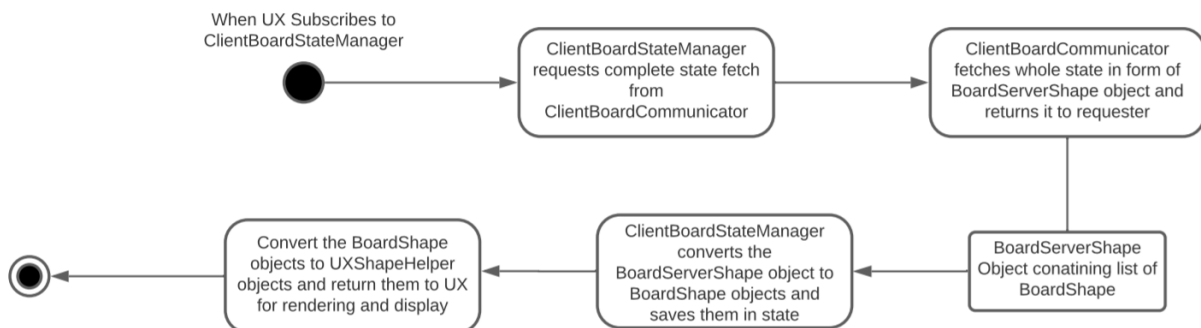
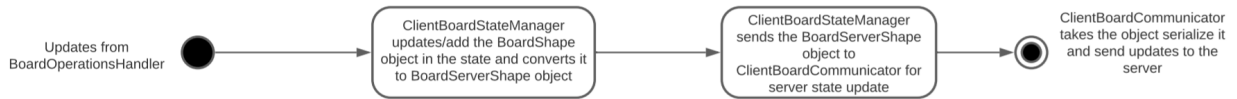


Figure 4. When UX Subscribes to ClientBoardStateManager

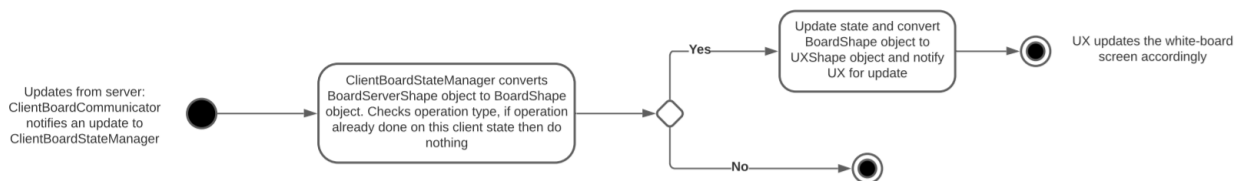
When UX subscribes to ClientBoardStateManager, the whole state which is saved at server is brought to this client via ClientBoardCommunicator which fetches the whole state. Then UX is provided with the sorted list of UXShapeHelper (sorted based on timestamp), so that the order of appearance of those shapes don't change. Along with state, a number which is the number of checkpoints which are saved on the server is also fetched, so that the current client has an option to switch to previous checkpoints as well.

---



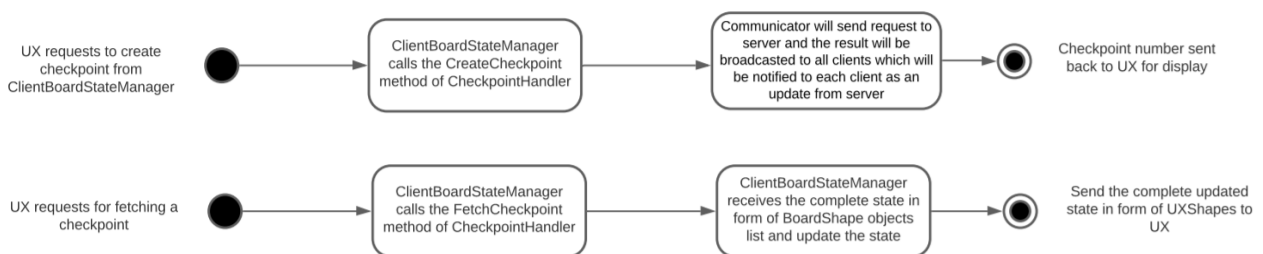
**Figure 5. On client-side operations**

On any client-side operation, first the state of the client is updated, then the update is sent to ClientBoardCommunicator to send it to server from where the update is broadcasted to all clients and that update is saved in server's state as well.



**Figure 6. When server update is received via ClientBoardCommunicator**

When an update is received from the server via ClientBoardCommunicator, firstly its checked whether it's the update from this client itself or not (since this client's updates would have been broadcasted). If yes, then UX already has that information, no need to notify them of this, since it's not a change. If not, then the state of the client is updated and UX is notified of the changes it has to make in the form of a list of UXShape.



**Figure 7. Interaction with CheckpointHandler**

Between CheckpointHandler and UX, ClientBoardStateManager acts as a middle-man. When a create checkpoint request comes to manager it forwards that request to CheckpointHandler which in turn creates a checkpoint and provides its number to manager, so that UX knows what number to display.

Similarly, when a fetch checkpoint request comes, CheckpointHandler fetches the whole state and provides it to the manager to replace its state with that. Then UX is notified and given the sorted list of UXShape to display (sorting based on timestamp).

## Server Side:

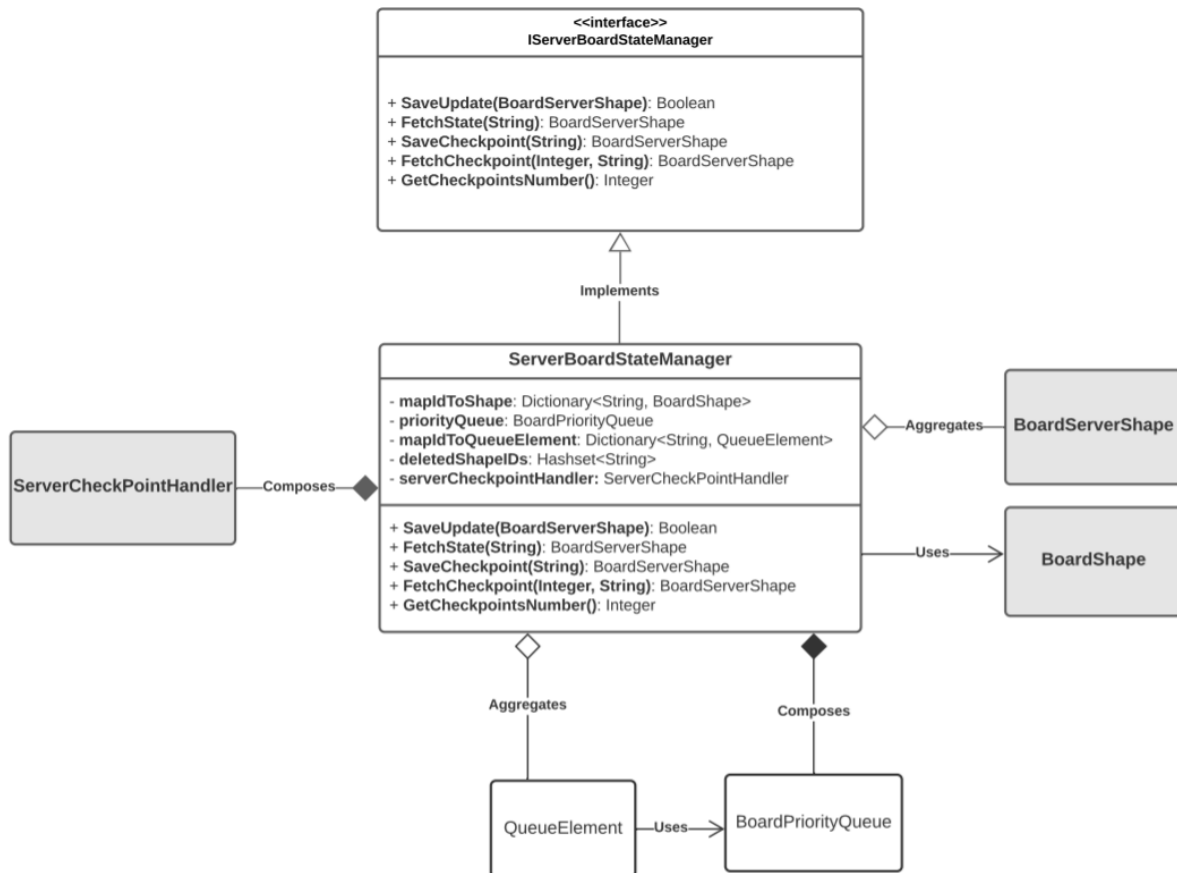


Figure 8. Server-side class diagram for Whiteboard State Manager: Including interfaces and state managing data structures

## How is the state managed?

At the server side, we will use the exact same pattern of data structures which we used at the client side. We will be using four data structures just like we used at Client side. However there will be no undo-redo stack like client because those are client-level operations.

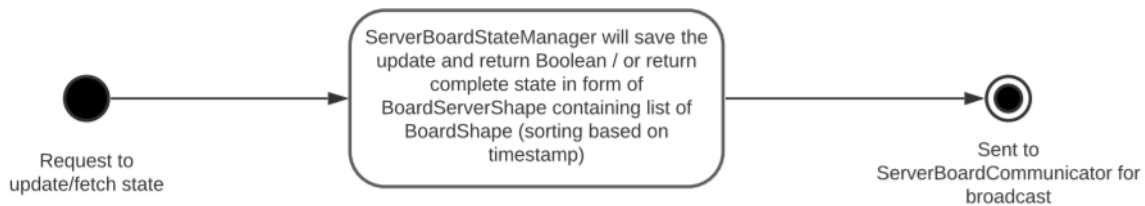
## Relationships with other components

This class will be composed by `ServerBoardCommunicator`. The main purpose of this class is to just provide utility to `ServerBoardCommunicator` to update the state, get the state, save checkpoint and fetch a checkpoint. The checkpoint management will be done by Checkpoint handler, the manager class is just acting as a middle man between the two, so that if a

checkpoint is fetched through it, it will be able to update server state with it as well.

This class will compose `ServerCheckpointHandler`, which in turn will be responsible for saving checkpoints in secondary storage at server's temp data.

### Activities at server-side



**Figure 9. When ServerBoardCommunicator updates state/fetches state (demand for state)**

When `ServerBoardCommunicator` calls `SaveUpdate()`, `ServerBoardStateManager` will save the update in its state. When `ServerBoardCommunicator` calls `FetchState()`, `ServerBoardStateManager` will create a `BoardServerShape` containing a sorted list (based on timestamp) of `BoardShape` and provide it to `ServerBoardCommunicator`.



**Figure 10. Interactions with ServerCheckpointHandler**

When `SaveCheckpoint()` is called, `ServerBoardCommunicator` will provide `ServerCheckpointHandler` a sorted list (based on timestamp) of `BoardShape` to `CheckpointHandler` to save a checkpoint. When `FetchCheckpoint()` is called, `CheckpointHandler` will return the saved checkpoint in the form of a sorted list of `BoardShape`, which will be used to update the state as well as broadcast to all clients as well.

## 9. Checkpoint Handler submodule (Owner: Chandan Srivastava)

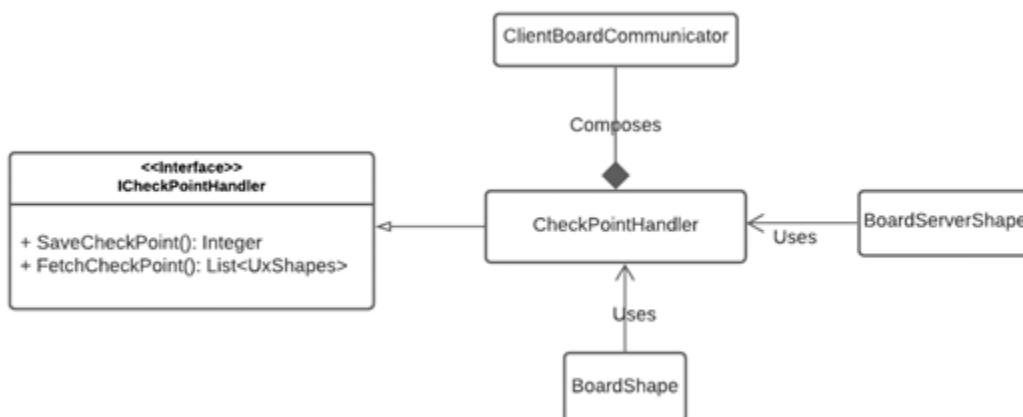
---

### 9.1 Objectives

- Server Side-
  - Write a function to store the state of the whiteboard in the file system and return the checkpoint number to the ClientBoardStateManager.
  - Write a function to fetch the stored state of the whiteboard in the file system and return the whole state to ClientBoardStateManager.
  - Build utility class Serialize which would contain functions to serialize and deserialize objects. It is used to serialize the board state so that it could store in (and later retrieved from) the filesystem for checkpoint.
    - Write a function serialize of the class Serialize to take an object (which is serializable) and convert it into a json from which we can completely recover the original object.
    - Write a function deSerialize of the class Serialize to take a json and return the corresponding object.
- Client Side-
  - Create BoardServerShapes object with SaveCheckpoint/ FetchCheckpoint operation and send it to ClientBoardCommunicator.

### 9.2 Design

#### Client Side:

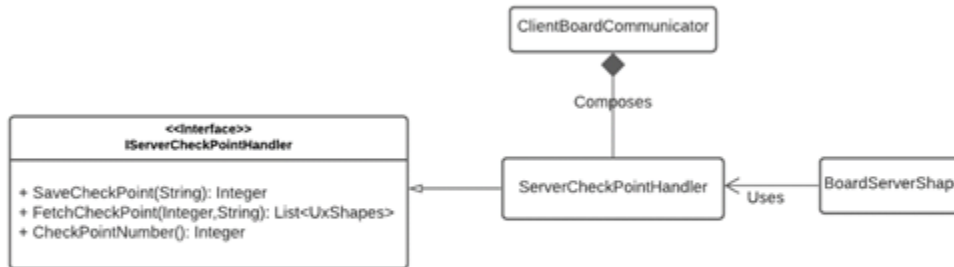


This class CheckPointHandler is composed by ClientBoardStateManager to do save and fetch checkpoint. CheckPointHandler creates BoardServerShapes object with SaveCheckpoint/ FetchCheckpoint operation and send it to ClientBoardCommunicator. ClientBoardCommunicator



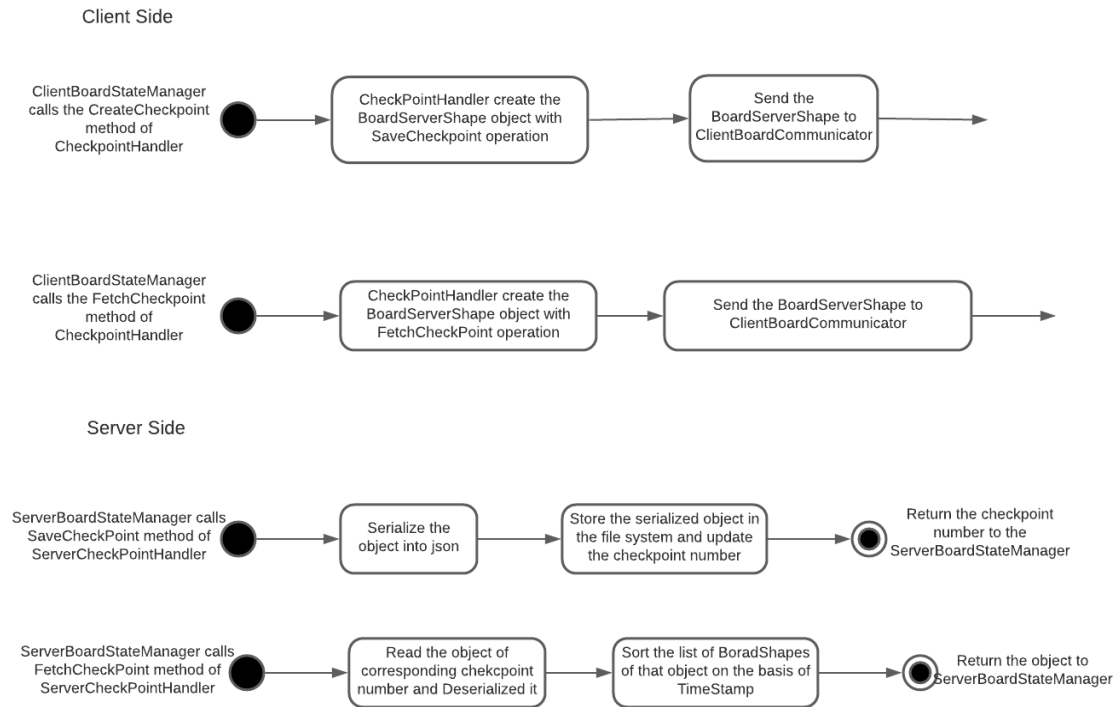
will do the required task and return the checkpointnumber/ board state to ClientBoardStateManager.

### Server Side:



This class **ServerCheckPointHandler** will be composed by **ServerStateBoardManager**. The purpose of this class is to provide utility to **ServerStateBoardManager** to save and fetch checkpoint. If the **ServerStateBoardManager** request to save checkpoint, it will save the checkpoint in the secondary storage at server's temp data and return the checkpoint number to the **ServerStateBoardManager**. And if the **ServerStateBoardManager** requests for fetching the checkpoint of given checkpoint number, it will return the state of the board to **ServerStateBoardManager**.

### 9.3 Activity diagram:



#### SaveCheckpoint at client side:

- ClientBoardStateManager requests the CheckPointHandler to save the checkpoint.
- CheckPointHandler Create BoardServerShapes object with SaveCheckpoint operation and send it to ClientBoardCommunicator.

#### FetchCheckpoint at client side:

- ClientBoardStateManager requests the CheckPointHandler to fetch checkpoint.
- CheckPointHandler Create BoardServerShapes object with FetchCheckpoint operation and send it to ClientBoardCommunicator.

#### Save CheckPoint at Server Side:

- ServerCheckPointHandler gets a request from ServerBoardStateManager to save the Checkpoint.
- Serialize the list of BoardShapes of that object into json.
- Store the serialized object in the file system and update the checkpoint number.
- Return the checkpoint number to the ServerBoardStateManager.

#### Fetch CheckPoint at Server Side:

- ServerCheckPointHandler gets a request from ServerBoardStateManager to fetch the Checkpoint of the given checkpoint number.
- ServerCheckPointHandler reads the file corresponding to that checkpoint number and deserializes it.
- Sort the list of Board Shapes of that object on the basis of TimeStamp.
- Return the sorted list of objects to ServerBoardStateManager.

#### **9.4 UserLevelHandler Class:**

- This class UserLevelHandler will be composed by BoardOperationsState. The purpose of this class is to provide utility to BoardOperationsState to find out whether the user have right to do the operation that he/she is wanted to do. There will be two user level 0 and 1. User with user level 1 can do the whiteboard operation on shapes which are owned by either user with user level 1 or user with user level 0. But users with user level 0 can perform the whiteboard operation on only those shapes which are owned by users with user level 0.

#### **9.5 Extended goal:**

- Provide a feature to take a screenshot of the current user screen (whiteboard) and store it in the server to include in session summary.
- Extend the number of user level from 2 (0 or 1) to n (0,1,2...n).

## 10. Board Server Communicator submodule (Owner: Gurunadh)

---

### 10.1 Overview:

There needs to be communication between client and server for broadcasting the changes to all other clients. Also to receive changes made by other clients through the server. The Networking Team will take care of transferring the XML strings between server and client. While the WhiteBoard team has to take care of all the implementation of all the logic. Thus there needs to be a communicator (BoardServerCommunicator & ServerBoardCommunicator) between these two modules to bridge the gap.

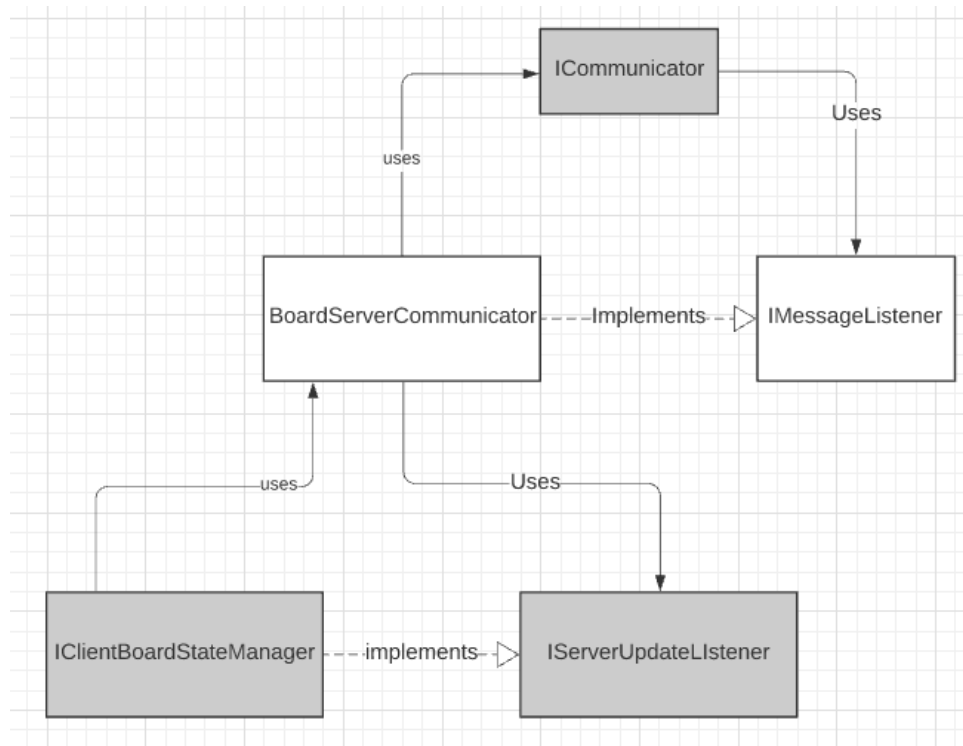
### 10.2 Objectives:

- Abstract out the complexities behind interconverting between BoardShape objects and the XML objects received from the Networking team.
- Serialize objects to send them from client side and server side.
- Deserialize and pass the BoardShape objects to client state manager and server state manager.
- Maintain a FIFO queue to hold the objects received at both client side and server side.

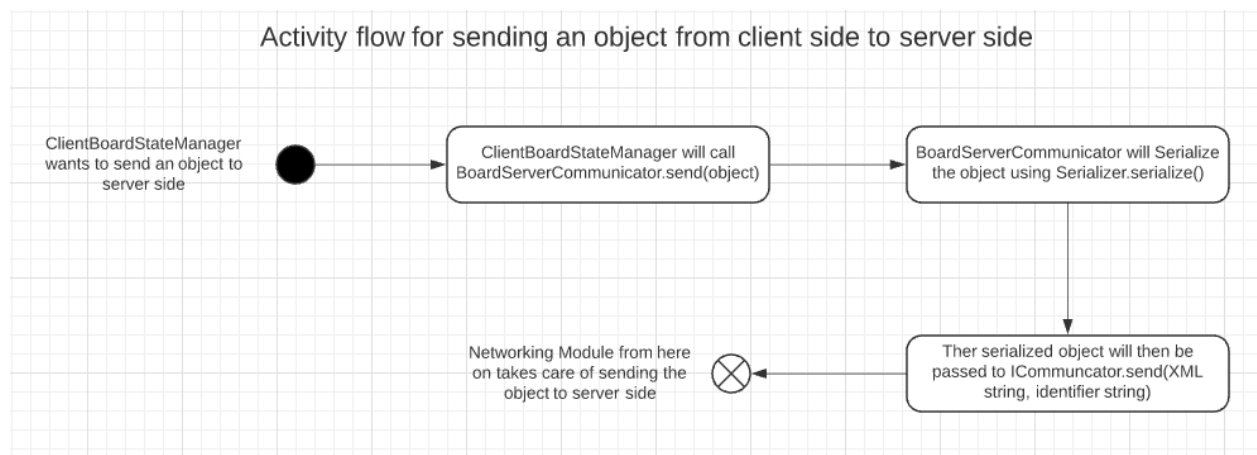
### 10.3 Design:

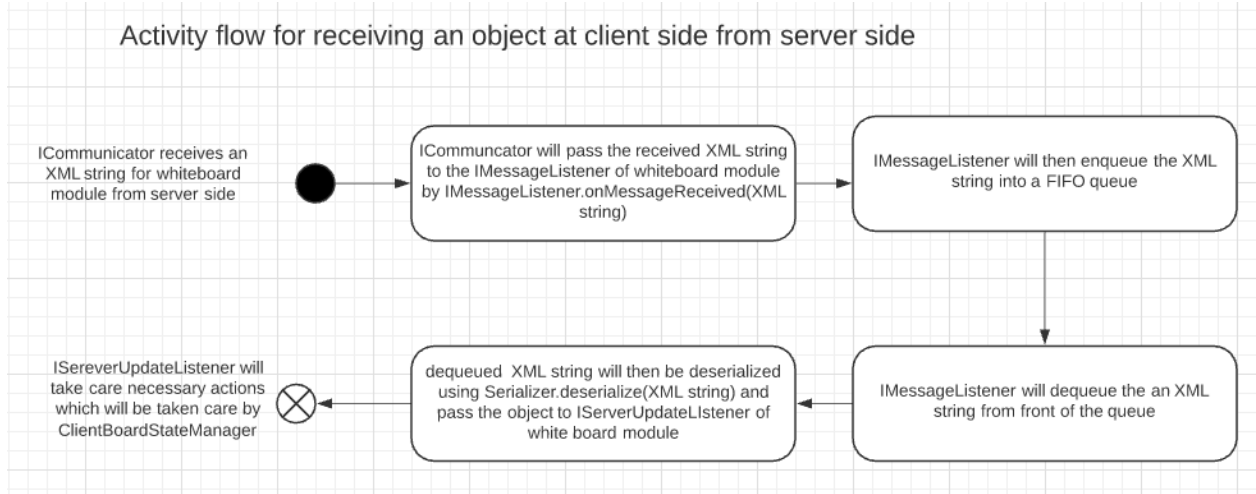
1. FIFO queue is chosen to store the objects that are received so that we give priority to the object that we received first
2. Publisher subscriber pattern is chosen to notify the client side state manager whenever we receive an object because it offers asynchronous message delivery. This pattern helps in keeping the system reliable even when features change. Publishers and Subscribers dont need to know each other in this pattern but in observer pattern observers are aware of the subject.

## 10.4 Class Diagram and Activity Diagram

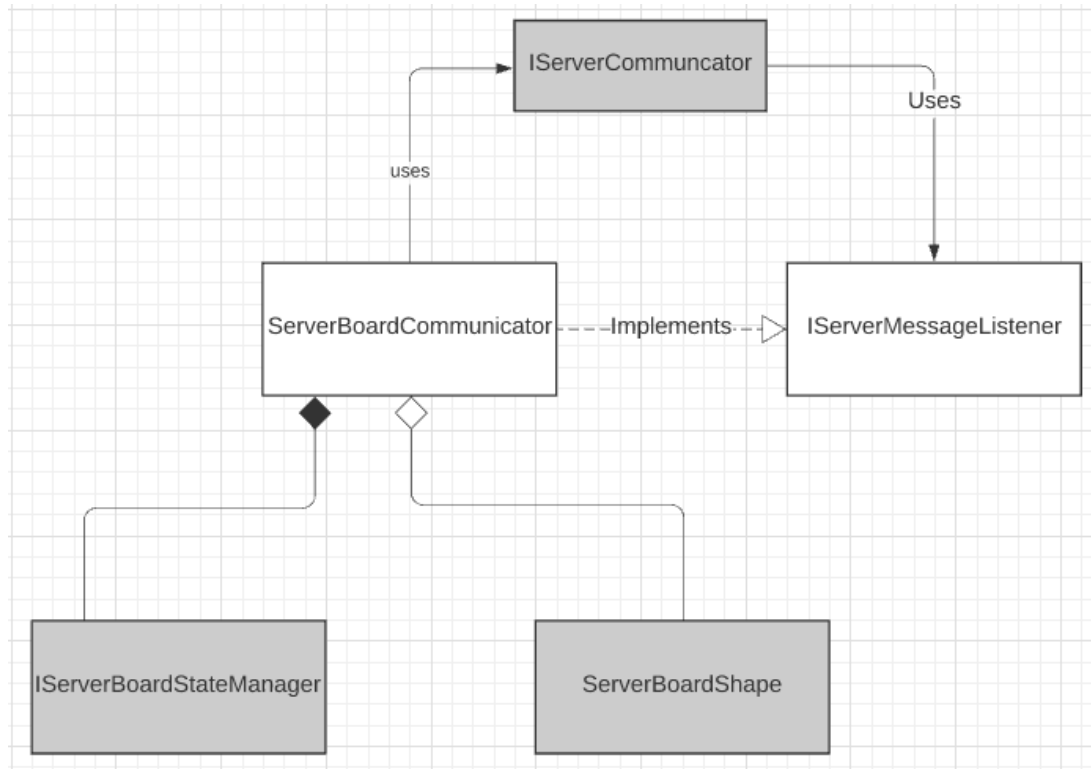


Class Diagram for BoardServerCommunicator at client side

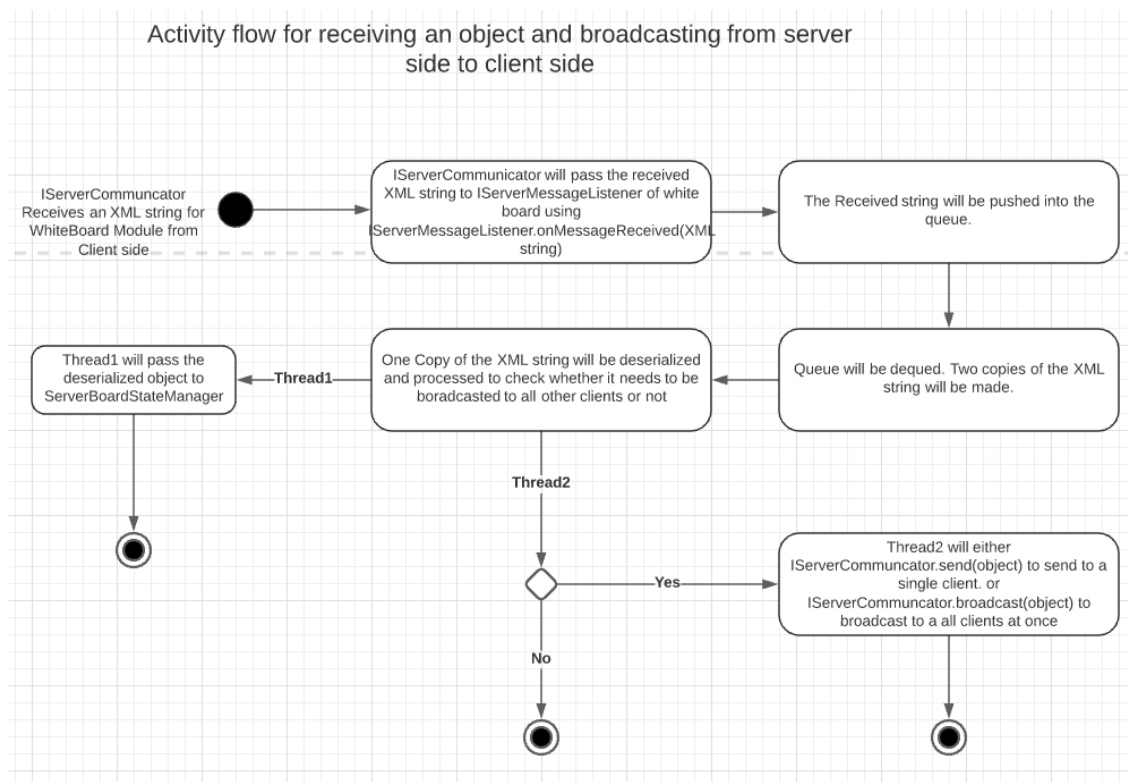




- The *ICommunicator* will be provided by the networking team which will use *IMessageListener.onMessageReceived()* to pass the XML objects received **from the server**. Received XML objects will be pushed to a queue.
- *BoardServerCommunicator* will use *ICommunicator.send()* to send serialized XML objects **to the server**.
- *IServerUpdateListener* will be subscribed to *BoardServerCommunicator* to receive deserialized objects. *BoardServerCommunicator* will call *IServerUpdateListener.onMessageReceived(deserialized object)*
- *ClientBoardStateManager* will take necessary actions according to the object received



**Class Diagram for ServerBoardCommunicator at Server side**



- The *IServerCommunicator* will be provided by the networking team which will use *IServerMessageListener.onMessageReceived()* to pass the XML objects received **from the client**. Received XML Objects will be pushed to a **queue**.
- *ServerBoardCommunicator* will use *IServerCommunicator.send()* to **send** serialized XML objects **to a particular client**.
- *ServerBoardCommunicator* will use *IServerCommunicator.Broadcast()* to **Broadcast** serialized XML objects **to all clients at once**.
- *ServerBoardCommunicator* will deque XML elements
- Two copies of XML objects will be made
- The XML element if needed will be broadcasted to all clients using *Thread1*
- *Thread2* will deserialize the XML object to *ServerBoardShape* object. Which then be passed to *IServerBoardStateManager* to save it to state.

### **Sending objects from client side to server side:**

1. *ClientSideStateManager* is going to use *BoardServerCommunicator.send(Object of any type)* to send an object from client side to server side.
2. The *BoardServerCommunicator.send()* will first serialize the object into XML String using the *Serializer* class provided by the Networking Team.
3. Now *CommunicationManager.getInstance()* will be called to get an instance of the communication manager which is the main contact for sending and receiving objects from and to the server side.
4. Once we get an instance of the communication manager, we can use *icomunicator.send(xml string, identifier string)* where identifier is to identify the module which is sending this xml string.
5. Thus from now on, networking team will take care of sending this xml string to the server side white board module

### **Code Structure:**

```
public interface BoardServerCommunicationManager
{
    public BoardServerCommunicator getInstance();
}
```

```
public interface BoardServerCommunicator
{
    public void Send(ShapeObject);
    public void subscribe(IServerUpdateListener UpdateListener);
}
```



}

### Receiving objects at client side from server side:

1. First of all, *IMessageListener* will be subscribed to *ICommunicator* using *ICommunicator.subscribe(IMessageListener object)* and *IServerUpdateListener* will be subscribed to *BoardServerCommunicator* using subscribe method provided.
2. Now whenever there is an object that is being sent from server side to the client, *ICommunicator* will invoke *IMessageListener.onMessageReceived(received object)* Using the *IMessageListener* object that was passed while subscribing to it
3. On receiving the object, it will be pushed to a queue which will contain the objects that are sent from server side
4. Whenever the queue has an object, *IMessageListener.onMessageReceived()* will **dequeue** and **deserialize** the xml string using the *Serializer.Deserialize()* to get an object of type that was passed at server side
5. Now *OnMessageReceived()* function will publish the received object to *IServerUpdateListener* using the object that was passed to it while subscribing.
6. Thus now the *ServerSideStateManager* will implement whatever they want to do on receiving the object from server side

### Code Structure:

```
public interface IServerUpdateListener
{
    public void onMessageReceived(ShapeObject);
}

public interface IMessageListener
{
    public void onMessageReceived(XML string);
}
```

### Receiving objects at server side from client side:

1. *IServerMessageListener* will be subscribed to *IServerCommunicator* by *IServerCommunicator.subscribe( IServerMessageListener )*
2. Thus whenever there is an object to our module from any client, *IServerMessageListener.onMessageReceived(XML string received)* will be called
3. Now all the received objects will be pushed into a **queue**
4. The *onMessageReceived()* function will **deque** the **xml** object

5. We keep 1 copy of this xml object
6. Now we deserialize the xml object and process the object
7. If the object can be broadcasted to all the clients we will spawn a **thread** to broadcast the saved copy of xml object to all clients
8. We will spawn another thread to call functions of the state manager according to the object.
9. Thus the server side state manager will then take care of necessary actions at server side with the deserialized object

#### Code Structure:

```
public interface IClientUpdateListener
{
    public void onMessageReceived(ShapeObject);
}

public interface IServerMessageListener
{
    public void onMessageReceived(XML string);
}
```

#### Sending / Broadcasting objects from server side to client(s):

1. Server side state manager will call *ServerBoardCommunicator.send(object of any type)*
2. Now Server Board communicator will first serialize the object using serializer provided by the networking team which returns an XML string
3. Server Board communicator will call *IServerCommunicator.getInstance* to get an instance of the *IServerCommunicator*
4. Server Board Communicator will the call *IServerCommunicator.send(XML string, destination)* to sending to a particular client or
5. It can call *IServerCommunicator.Broadcast(XML string)* to broadcast the serialized XML to all the clients at once.
6. Thus the networking module will take care of sending the XML strings to clients.

#### Code Structure:

```
public interface ServerBoardCommunicationManager
{
    public ServerBoardCommunicator getInstance();
}
```

```
public interface ServerBoardCommunicator
{
    public void Send(Object);
    public void subscribe(IClientUpdateListener UpdateListener);
}
```

## 11. Screenshare submodule (Owner: Neeraj Patil)

---

### 11.1 Overview:

A Screen Sharing module is very important in a discussion board application. It enables the users of the application to communicate and express their thoughts and ideas more effectively and also increases their productivity. This module will have two parts i.e. the Server part and the Client part

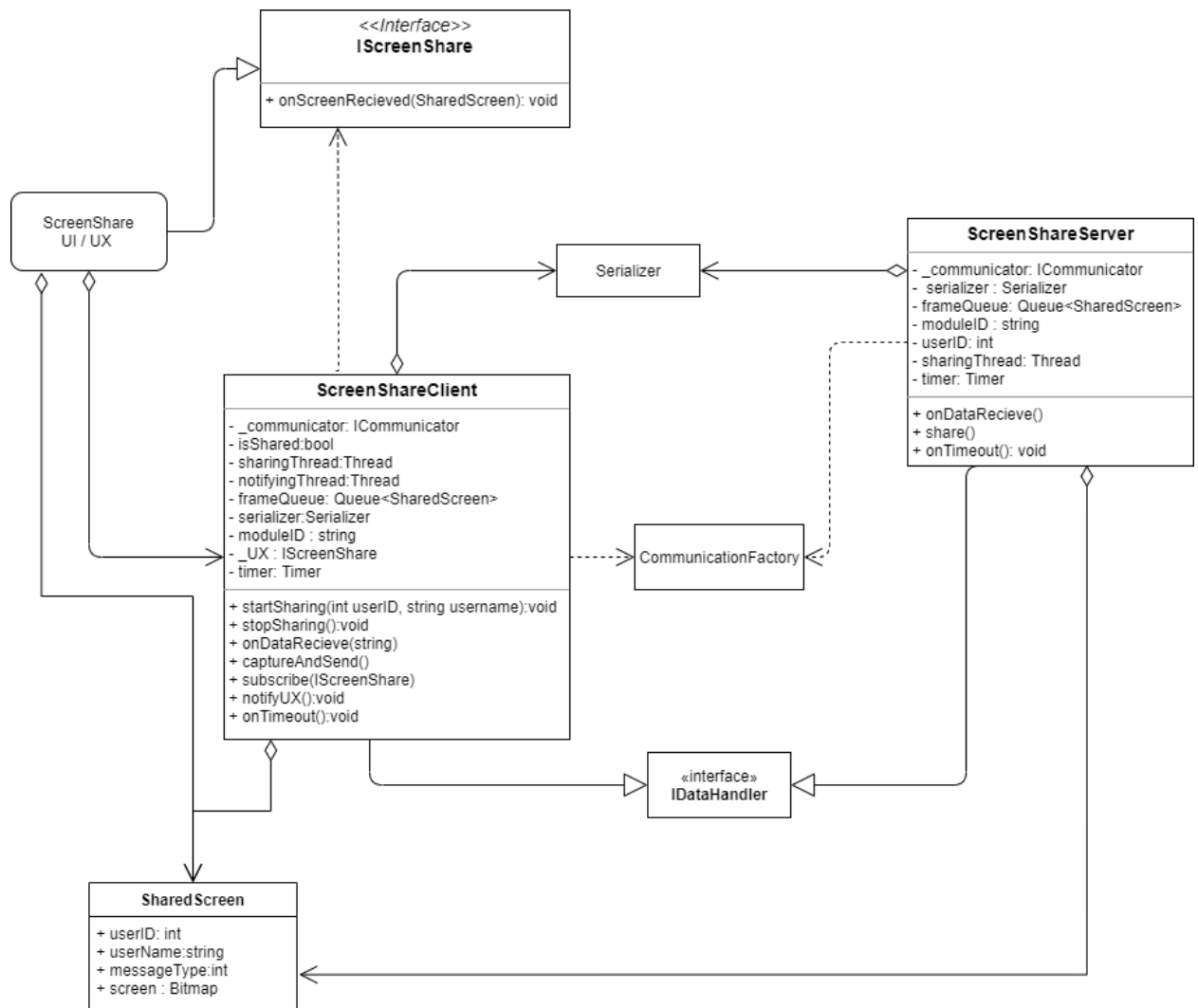
### 11.2 Objectives:

- **Phase 1:**
  - To create a simple screen sharing module which shares the entire screen for the discussion board application.
- **Phase2:**
  - If possible, then implement the option to share a specific window or a tab of a browser.
  - If possible, give the user an option to select the resolution in which the screen will be shared. E.g.: 240p, 360p, 480p, 720p.
  - If possible let the user choose the FPS (frames per second) while sharing the screen.

#### **Important Aspects for Phase 1:**

- The resolution at which the shared screen will be received by the clients is 480p. The pixel resolution for 480p is 640X480. Due to this, the size of one frame of this resolution is 1.17MB. Also all the desktop and laptop monitors support this resolution.
- Choosing a lower resolution might hinder the experience of the clients as the screen might not be clear for viewing small fonts or minute details.
- For higher resolution, such as 720p, the frame size increases to almost 3.6MB. Due to large frame size, all the higher resolutions are not being used in phase 1.
- For the frequency of frames in screen sharing, the minimum frame rate required for smooth user experience is 24fps. So I will start with 24fps and then if the networking team can handle that efficiently, we will experiment with it and get it to an optimum for both smoothness and performance.

### 11.3 UML diagram:



### 11.4 Design Analysis:

- Between the UX and the Screen Share modules in the client side, the design pattern which is used is the observer pattern as there is only one observer (UX) waiting for the notification from the Screen Sharing. This is implemented using the interface 'IScreenShare'.
- Similarly, between the screen share modules and the Networking module, publisher subscriber design pattern is used. This is because there will be many subscribers for the Networking module apart from the Screen Sharing module. This design pattern is implemented using the 'IDataHandler' interface.

#### Class Details:

- ScreenShareUI/UX:**

- This class will be created by the UX team. This class will inherit the interface 'IScreenShare'. The constructor of this class will create an instance of the 'ScreenShareClient' and will subscribe to it using the 'subscribe' method.
- Whenever the 'Screen Share' button is clicked, it will use the instance of the 'ScreenShareClient' class to call the method 'startSharing'.
- As it uses the 'Observer' pattern, it uses the method 'onScreenRecieved' to get the bitmap from the 'ScreenShareClient' and then renders it on the UI.

- **ScreenShareClient:**

- The constructor of this class creates the instances of the following classes:
  - serializer using the 'Serializer' class.
  - \_communicator using the 'CommunicationFactory'
- It will also subscribe to the Communicator for the notifications inside the constructor.
- In the 'startSharing' method, a thread will be created which will use the 'captureAndSend' method to implement the logic of capturing the screen and broadcasting it. It will use the serializer to serialize the data before sending it to the server using the communicator.
- In the 'stopSharing' method, the sharing thread will be destroyed and a stop sharing signal will be sent using the communicator.
- The notifying thread will be created in the constructor itself. The goal of this thread is to notify the UX about the received screen. This thread will be blocked when the 'frameQueue' is empty. It will use the 'notifyUX()' method.
- The 'onDataRecieve()' method will be used by the networking team to notify about the shared screen which is received. This method will take the screen and push it in the 'frameQueue'.

- **ScreenShareServer:**

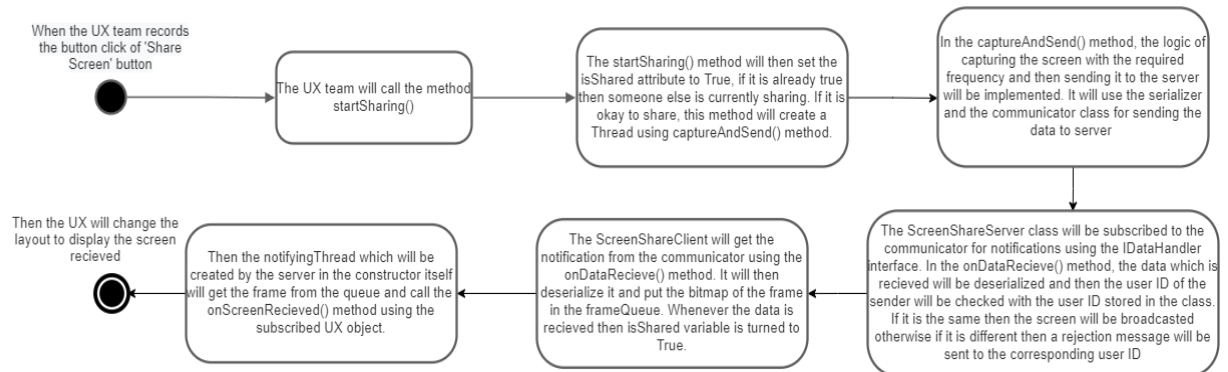
- The constructor of this class will create an instance of the communicator class using the 'CommunicationFactory', it will also create an instance of the serializer.
- It will maintain a 'frameQueue' to organize the incoming frames. This 'frameQueue' will be filled by the 'onDataRecieve' method.
- A sharing thread will be created by the constructor which will use the share method to send data to the clients. This method will be blocked when the 'frameQueue' is empty.
- The 'userID' attribute will be used to check whether some client is already sharing the screen or not. If some other client is sharing the screen, then a rejection message will be sent to the client who is trying to share the screen.

### Disconnection Issues:

- For the disconnection issues, I have used a timer in the 'ScreenShareClient' and 'ScreenShareServer' classes.
- This timer will start when anyone starts sharing and will be reset whenever an update is received.
- Whenever updates stop for some amount of time (e.g. 1minute), this timer will timeout.
- **When Client Listening for the screen gets disconnected:**
  - The timer in the 'ScreenShareClient' will timeout as no updates will be received and then the onTimeout() method will be called. This method will use the notifyUX() method to change the layout of the UX accordingly.
- **When the Client Sharing the screen gets disconnected:**
  - The client which is sharing the screen has its own listener. So when the listener will stop getting updates, the timer will timeout which will call the method onTimeout(). This method will check whether the client is the sharer. If it is, then it will abort the sharingThread and will use 'notifyUX()' method to notify the UX team.
  - There is a timer on the server side as well. So if the updates stop coming, the timeout event will occur and call the 'onTimeout()' method which will make the frameQueue empty and will be open to sharing for others.
- **When the server gets disconnected:**
  - If the server gets disconnected timeout will happen for all the clients as well as the server. So the layout for the clients will change and the server will come to its initial state i.e. the frame queue will become empty and it will be open for sharing when it is connected again.

## 11.5 Activity Diagram:

### 1. Start Sharing the Screen



### 2. Stop Sharing the screen

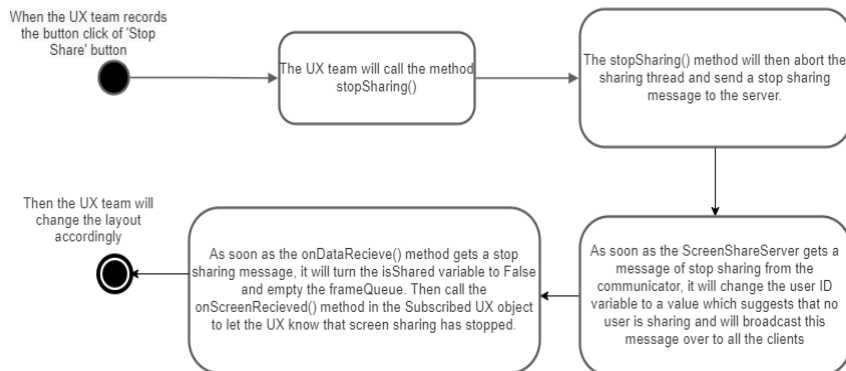


Figure: Activity diagram denoting the workflow

## 12. Interfaces

---

The following section lists down all the public interfaces exposed which are to be used by other modules:

### 1. IWhiteBoardOperationHandler

```
public interface IWhiteBoardOperationHandler
{
    /// <summary>
    /// Creates Ellipse/Circle.
    /// </summary>
    /// <param name="start"> Coordinate of mouse down event. </param>
    /// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
    /// <param name="strokeWidth"> Shape boundary stroke width. </param>
    /// <param name="strokeColor"> Color of shape boundary stroke. </param>
    /// <param name="shapeId"> Id of the shape. Null if shape creation just started. </param>
    /// <param name="shapeComp"> indicative of a mouse up event. </param>
    /// <returns> List of UXShapes for UX to render. </returns>
    List<UXShape> CreateEllipse(Coordinate start, Coordinate end, float strokeWidth, BoardColor
strokeColor, string shapeId = null, bool shapeComp = false);

    /// <summary>
    /// Creates Rectangle/Square.
    /// </summary>
    /// <param name="start"> Coordinate of mouse down event. </param>
    /// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
    /// <param name="strokeWidth"> Shape boundary stroke width. </param>
    /// <param name="strokeColor"> Color of shape boundary stroke. </param>
    /// <param name="shapeId"> Id of the shape. Null if shape creation just started. </param>
    /// <param name="shapeComp"> indicative of a mouse up event. </param>
    /// <returns> List of UXShapes for UX to render. </returns>
    List<UXShape> CreateRectangle(Coordinate start, Coordinate end, float strokeWidth,
BoardColor strokeColor, string shapeId = null, bool shapeComp = false);

    /// <summary>
    /// Creates Polyline.
    /// </summary>
    /// <param name="start"> Coordinate of mouse down event. </param>
    /// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
    /// <param name="strokeWidth"> Shape boundary stroke width. </param>
    /// <param name="strokeColor"> Color of shape boundary stroke. </param>
    /// <param name="shapeId"> Id of the shape. Null if shape creation just started. </param>
    /// <param name="shapeComp"> indicative of a mouse up event. </param>
    /// <returns> List of UXShapes for UX to render. </returns>
```



```

List<UXShape> CreatePolyline(Coordinate start, Coordinate end, float strokeWidth,
BoardColor strokeColor, string shapeId = null, bool shapeComp = false);

/// <summary>
/// Creates straight line.
/// </summary>
/// <param name="start"> Coordinate of mouse down event. </param>
/// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
/// <param name="strokeWidth"> Shape boundary stroke width. </param>
/// <param name="strokeColor"> Color of shape boundary stroke. </param>
/// <param name="shapeId"> Id of the shape. Null if shape creation just started. </param>
/// <param name="shapeComp"> indicative of a mouse up event. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> CreateLine(Coordinate start, Coordinate end, float strokeWidth, BoardColor
strokeColor, string shapeId = null, bool shapeComp = false);

/// <summary>
/// Translates the shape with a given shape ID.
/// </summary>
/// <param name="start"> Coordinate of mouse down event. </param>
/// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
/// <param name="shapeId"> Id of the shape. </param>
/// <param name="shapeComp"> Indicative of a mouse up event. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> TranslateShape(Coordinate start, Coordinate end, string shapeId, bool
shapeComp = false);

/// <summary>
/// Rotates the shape with a given shape ID.
/// </summary>
/// <param name="start"> Coordinate of mouse down event. </param>
/// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
/// <param name="shapeId"> Id of the shape. </param>
/// <param name="shapeComp"> Indicative of a mouse up event. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> RotateShape(Coordinate start, Coordinate end, string shapeId, bool shapeComp
= false);

/// <summary>
/// Resizes the shape with a given shape ID.
/// </summary>
/// <param name="start"> Coordinate of mouse down event. </param>
/// <param name="end"> Current coordinate to display real-time shape creation before/at
mouse up event. </param>
/// <param name="shapeId"> Id of the shape. </param>
/// <param name="shapeComp"> Indicative of a mouse up event. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> ResizeShape(Coordinate start, Coordinate end, string shapeId, DragPos
dragpos, bool shapeComp = false);

```

```

/// <summary>
/// Changes the thickness of the shape outline stroke.
/// </summary>
/// <param name="strokeWidth"> Stroke Thickness. </param>
/// <param name="shapeId">Id of the shape. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> ChangeStrokeWidth(float strokeWidth, string shapeId);

/// <summary>
/// Changes the Stroke Color of the shape outline.
/// </summary>
/// <param name="strokeColor"> Stroke Color. </param>
/// <param name="shapeId">Id of the shape. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> ChangeStrokeColor(BoardColor strokeColor, string shapeId);

/// <summary>
/// Changes the Fill Color of the shape.
/// </summary>
/// <param name="shapeFill"> Shape Fill Color. </param>
/// <param name="shapeId">Id of the shape. </param>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> ChangeShapeFill(BoardColor shapeFill, string shapeId);

/// <summary>
/// Performs Undo.
/// </summary>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> Undo();

/// <summary>
/// Performs Redo.
/// </summary>
/// <returns> List of UXShapes for UX to render. </returns>
List<UXShape> Redo();

/// <summary>
/// Gets owner of the shape with a shape Id.
/// </summary>
/// <param name="shapeId"> Id of the shape. </param>
/// <returns> User Name. </returns>
string GetUserName(string shapeId);

/// <summary>
/// Switches WhiteBoard state from active to inactive and vice-versa.
/// </summary>
/// <returns> Denotes successful state switch. </returns>
bool SwitchState();

/// <summary>
/// Delete shape with given shape Id.

```

```

    /// </summary>
    /// <param name="shapeId">Id of the shape.</param>
    /// <returns></returns>
    List<UXShape> DeleteShape(string shapeId);
}

```

---

## 2. IClientBoardStateManager

```

public interface IClientBoardStateManager
{
    /// <summary>
    /// Static method to get the single created instance of ClientBoardStateManager.
    /// </summary>
    static ClientBoardStateManager Instance { get; }

    /// <summary>
    /// Initializes state managers attributes.
    /// </summary>
    void Start();

    /// <summary>
    /// Subscribes to notifications from ClientBoardStateManager to get updates.
    /// </summary>
    /// <param name="listener">The subscriber. </param>
    /// <param name="identifier">The identifier of the subscriber. </param>
    void Subscribe(IClientBoardStateListener listener, string identifier);

    /// <summary>
    /// Creates and saves a checkpoint.
    /// </summary>
    void SaveCheckpoint();

    /// <summary>
    /// Fetches the checkpoint from the server and updates the current state.
    /// </summary>
    /// <param name="checkpointNumber">The identifier/number of the checkpoint which
    needs to fetched.</param>
    void FetchCheckpoint(int checkpointNumber);

    /// <summary>
    /// Sets the current user id.
    /// </summary>
    /// <param name="userId">user Id of the current user.</param>
    void SetUser(string userId);

    /// <summary>
    /// Clears the whiteboard to fresh.
    /// </summary>
    /// <returns>Boolean to freeze whiteboard or not.</returns>
    bool ClearWhiteBoard();
}

```

```
}
```

---

### 3. IMessageListener and IServerMessageListener

```
public interface IMessageListener
{
    /// <summary>
    ///     Receives XML strings from ICommunicator
    /// </summary>
    /// <param name="data"> XML string from communicator </param>
    void OnMessageReceived(string data);
}

public interface IServerMessageListener
{
    /// <summary>
    ///     Receives XML strings from IServerCommunicator
    /// </summary>
    /// <param name="data">
    ///     Serialized Object in form of
    ///     XML string that was sent from some client
    /// </param>
    void OnMessageReceived(string data);
}
```

### 4. IScreenReceived

```
public interface IScreenReceived
{
    public void onScreenReceived(Bitmap map);
}
```

## 13. Future Scope

---

- Provide a feature to take a screenshot of the current user screen (whiteboard) and store it in the server to include in the session summary.
- Extend the number of user levels from 2 (0 or 1) to n (0,1,2...n).
- If possible, give the user an option to select the resolution in which the screen will be shared. E.g.: 240p, 360p, 480p, 720p.
- If possible let the user choose the FPS (frames per second) while sharing the screen.

## 14. Features Delivered

1. **The Whiteboard module** supporting the following:
  - a. **Creation of Shapes.**
    - i. Rectangle (Square)
    - ii. Ellipse (Circle)
    - iii. Line
    - iv. Polyline (Free-hand curves)
  - b. **Operation on Shapes (Not including Polyline):**
    - i. Translation
    - ii. Resizing
    - iii. Rotation
    - iv. Deletion
    - v. Shape border thickness/color change
    - vi. Shape fill-color change
  - c. **Erasing of polyline.** (A Stroke of eraser on polyline deletes the complete polyline)
  - d. **Active/Inactive mode for whiteboard.**
  - e. **User Level switches** influencing which user can modify a shape.
  - f. **Saving, creating and restoring multiple checkpoints.**
  - g. **Auto-fetch the server-state** on new client arrival.
  - h. **Undo-Redo**
  - i. **Getting user-id on providing id of the shape.**
  - j. **Clearing whiteboard**