Meet.me

Module Specifications: WhiteBoard and ScreenShare Module

Manas Sharma
Team Lead(WhiteBoard and Screen Sharing)

Contents

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 the *Convertor* interface to convert the non-serializable *BoardShapeObject* to a serializable *BoardServerObject* which will be used for broadcasting over the server.

➔ **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.

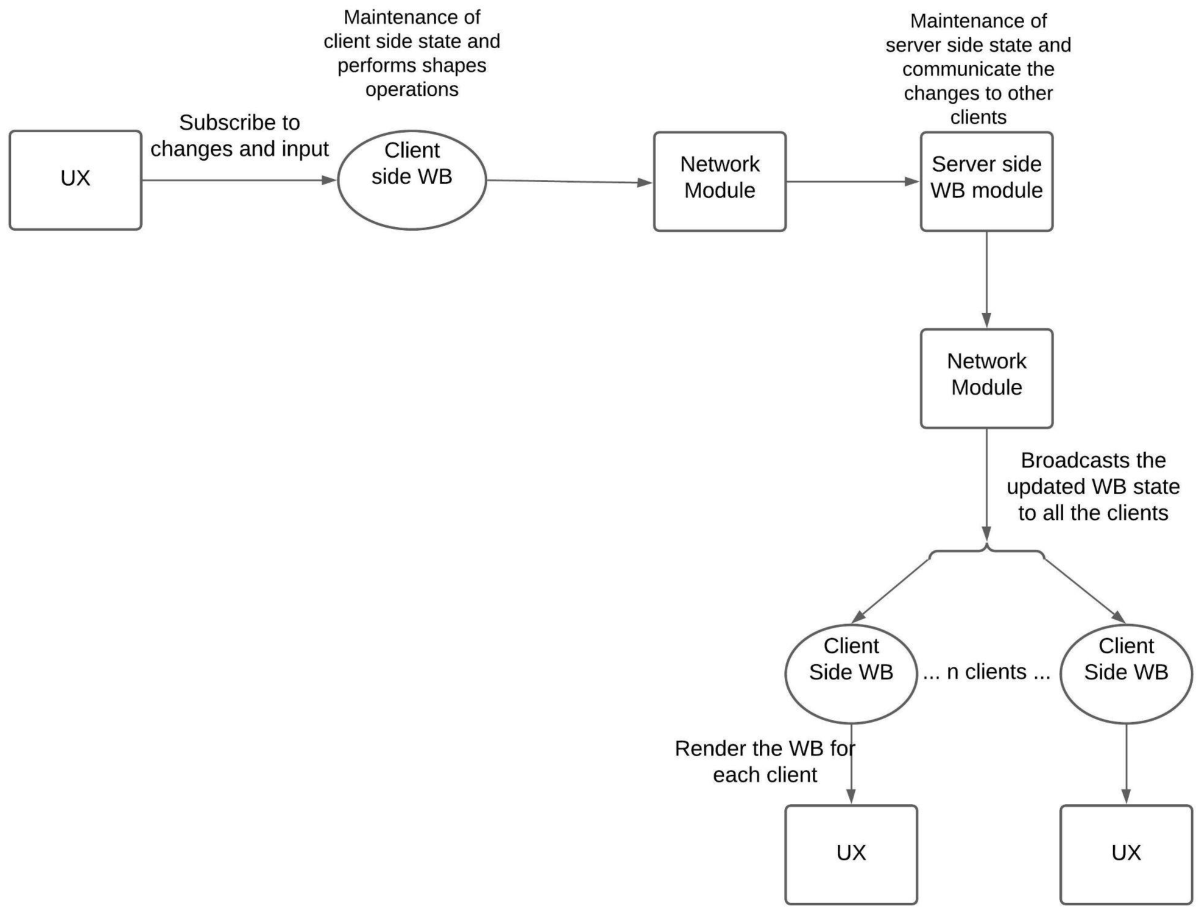

Maintenance of client side state and performs shapes operations
Maintenance of server side state and communicate the changes to other clients
UX
Subscribe to changes and input
Client side WB
Network Module
Server side WB module
Network Module
Broadcasts the updated WB state to all the clients
Client Side WB
... n clients ...
Client Side WB
Render the WB for each client
UX
UX

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.

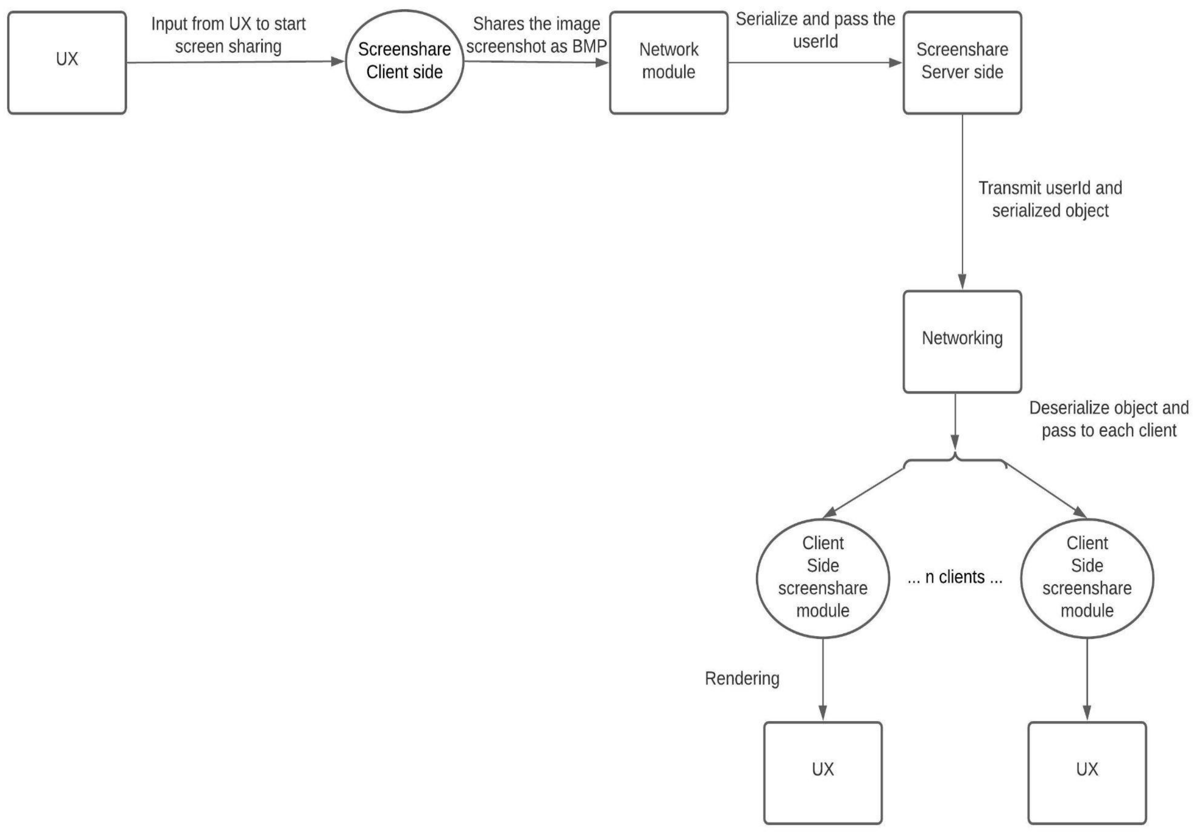

UX
Input from UX to start screen sharing
Screenshare Client side
Shares the image screenshot as BMP
Network module
Serialize and pass the userId
Screenshare Server side
Transmit userId and serialized object
Networking
Deserialize object and pass to each client
Client Side screenshare module
... n clients ...
Client Side screenshare module
Rendering
UX
UX

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.

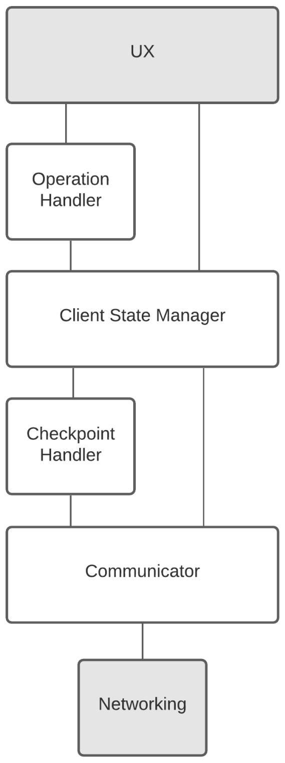


Figure 3. Intra-module dependency(client)

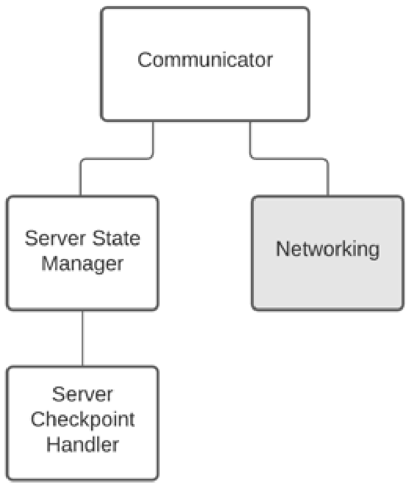


Figure 4. Intra-module dependency(server)

6. UML Diagram

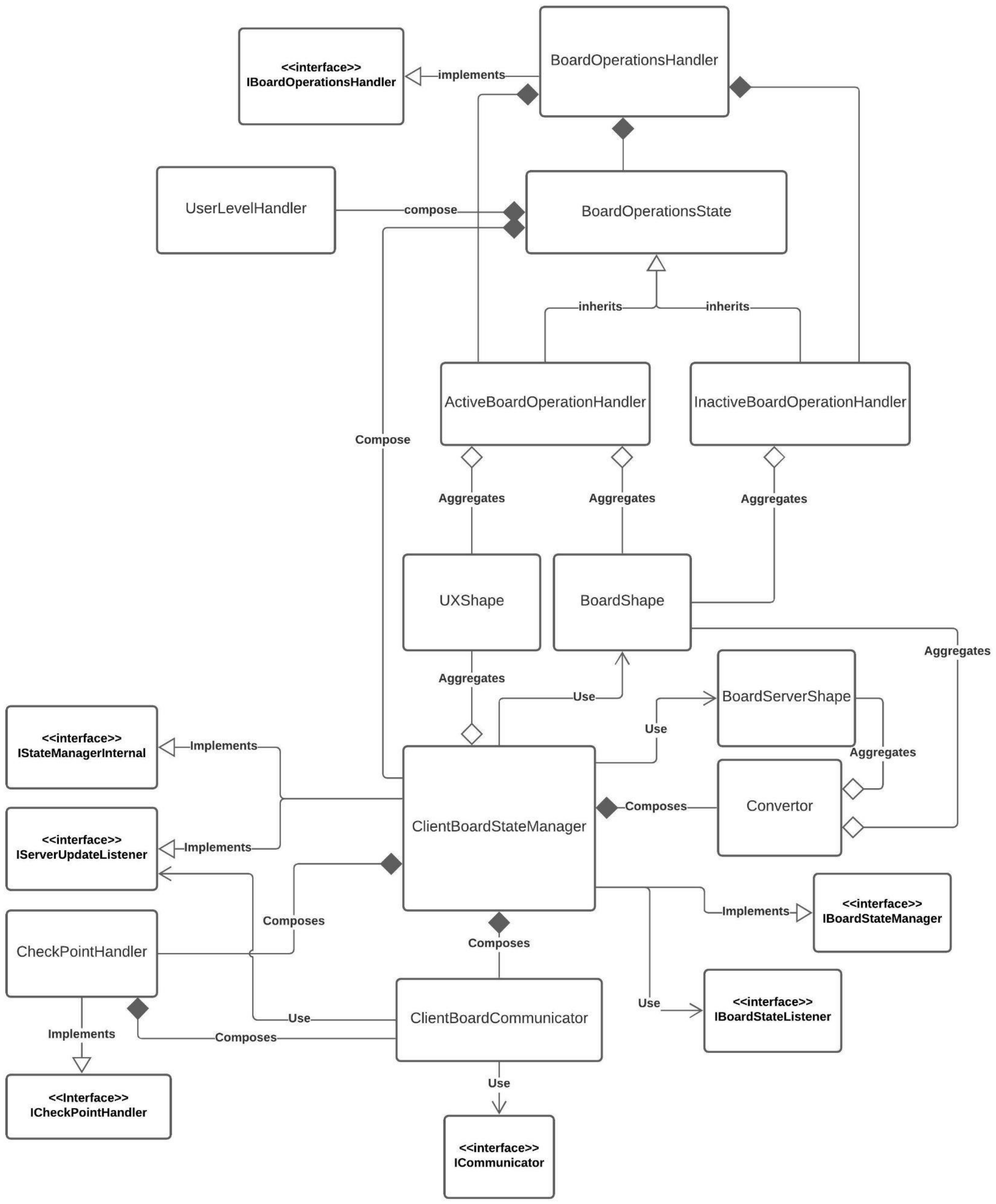


Figure 5: UML diagram for client side operations for Whiteboard module

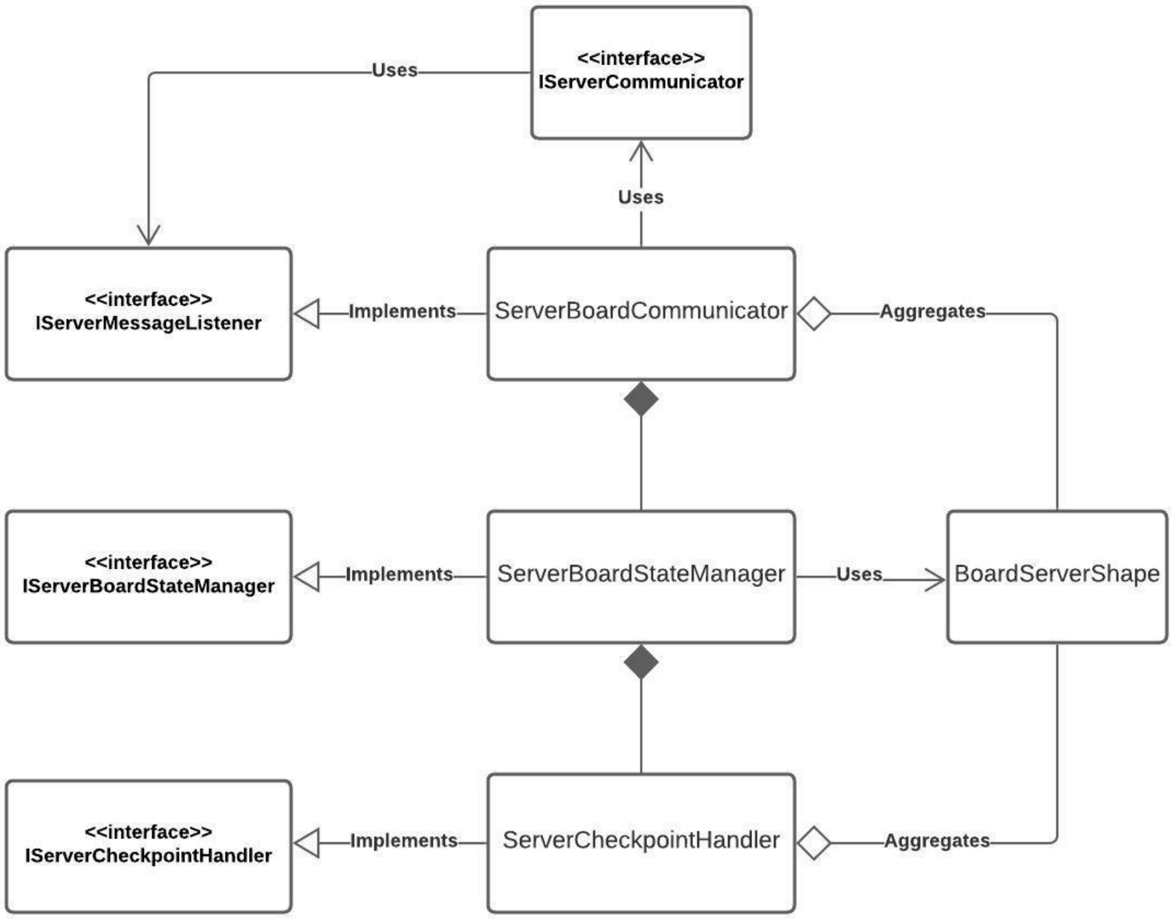


Figure 6: UML diagram for server side operations for Whiteboard module

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.

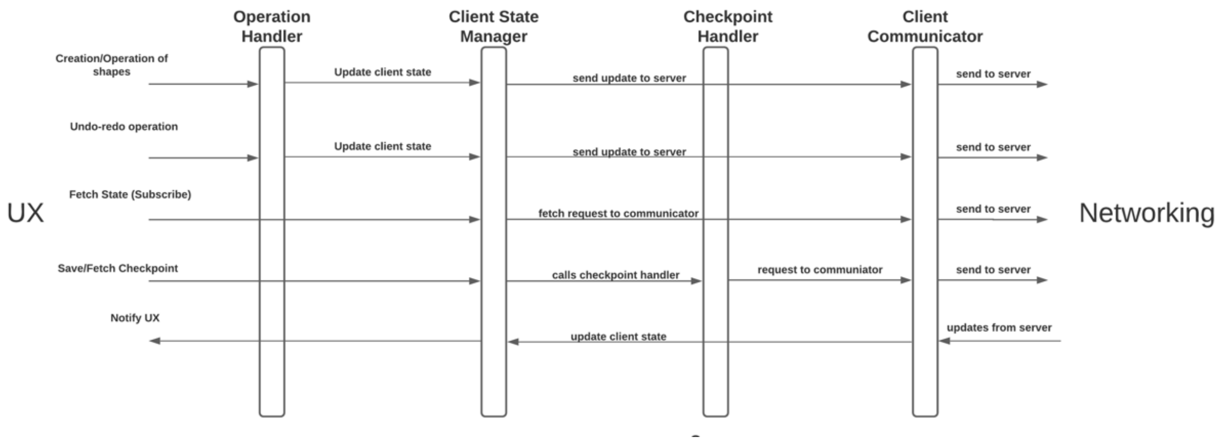


Figure 7: Activity flow diagram for Whiteboard(client side)

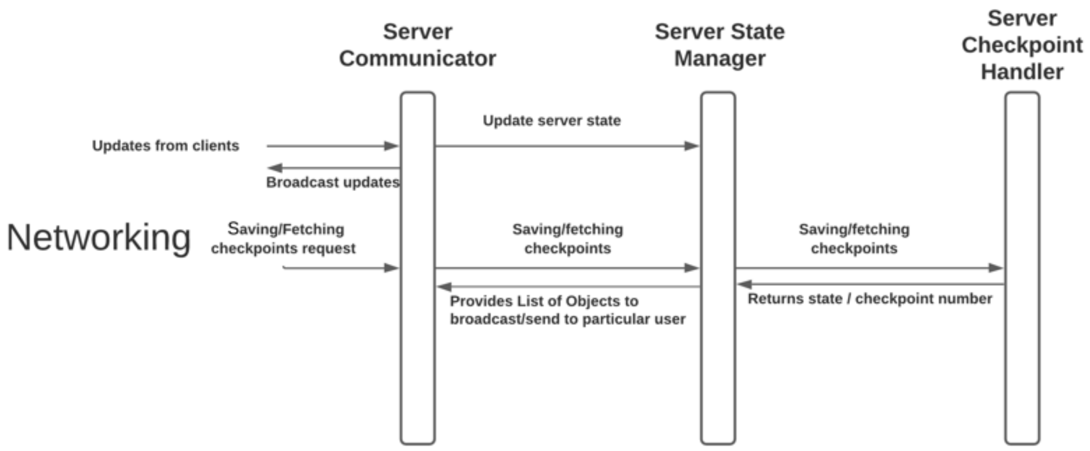

Server Communicator
Server State Manager
Server Checkpoint Handler
Networking
Updates from clients
Broadcast updates
Update server state
Saving/Fetching checkpoints request
Saving/fetching checkpoints
Saving/fetching checkpoints
Provides List of Objects to broadcast/send to particular user
Returns state / checkpoint number

Figure 8 :Activity diagram for Whiteboard(Server side)

7. Whiteboard Operations submodule

7.1 Objectives

Shape Operations and Creation

- Provide an interface to the UX module for creation, deletion, undo, redo, or performing updates of objects on the UX for instance resizing and changing shape fill, freezing/unfreezing the whiteboard, etc, along with additional methods to get users 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.
- While the creation of shape or shape Operation, Storing the previous shape for providing updates to the client for real-time display of shape before the mouse up event takes place and the final object is decided.

Freezing and Unfreezing Functionality

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

Convertor for WhiteBoard Objects

- Implement a Class for to and from conversion for converting the non-serializable Client-Side BoardShape Object to a serializable BoardServerObject with similar information for sending to the server for Broadcasting.

7.2 Design

1. **Overall design**

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

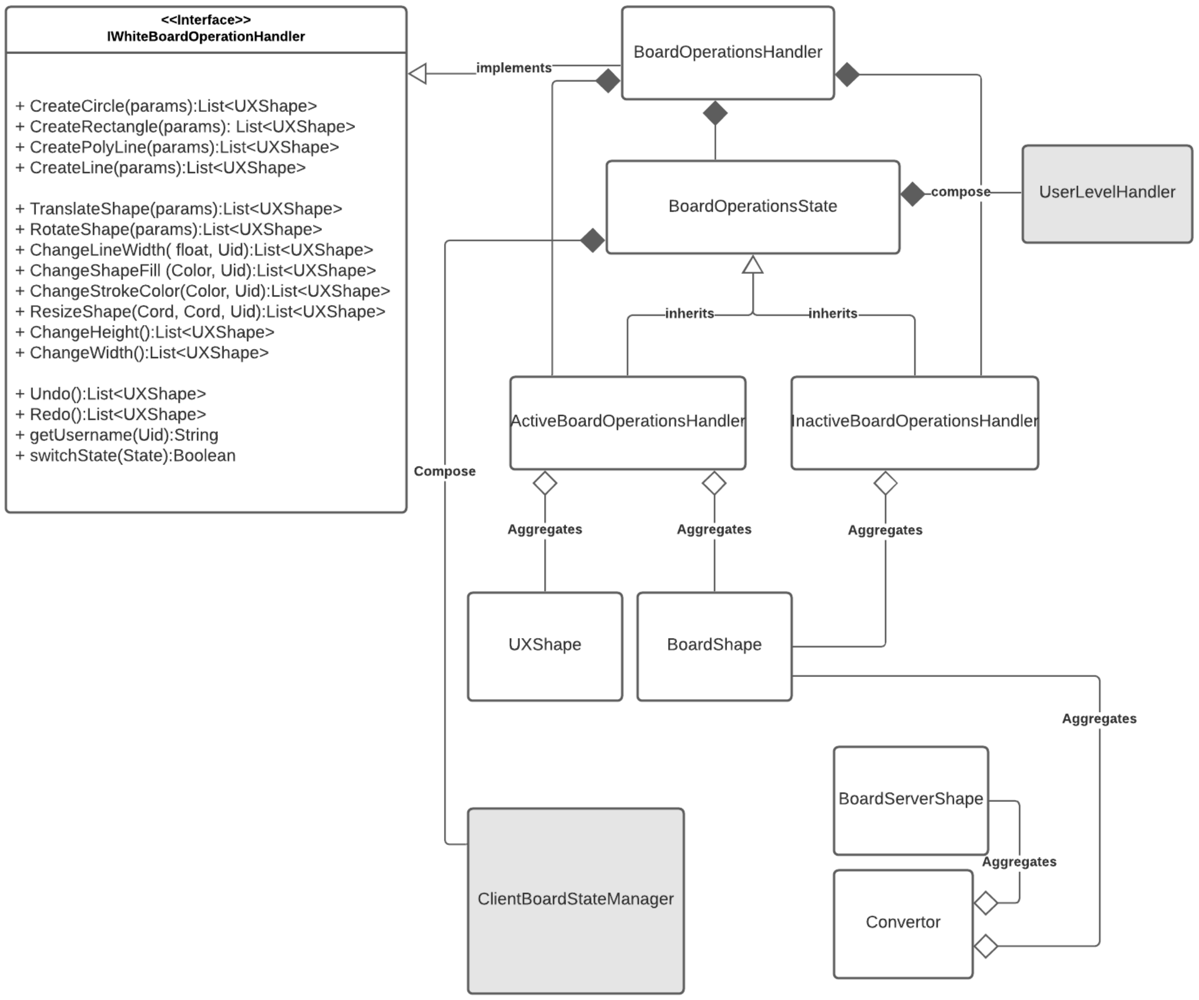
1. Inactive - The client won't be able to make any changes himself, but will be able to see the changes of other people on his whiteboard.

2. Active - The client can actively participate in making edits to the shapes on the whiteboard.

Based on this, the Handler Class will call appropriate methods to update the object and correspondingly update the State Management at the client before broadcasting the changes to the other clients.
When it's time to send the Shapes to the Server, this Shape needs to be serialized, since System.Windows.Shapes Class is not directly serializable. This to and from the conversion is done at the Client Side, when the Object leaves the client to go to the server and when the object comes back to the client on the server.

2. UML Diagram

Note: the grey-colored classes belong to other team members.



<<Interface>>
IWhiteBoardOperationHandler
+ CreateCircle(params):List<UXShape>
+ CreateRectangle(params): List<UXShape>
+ CreatePolyLine(params):List<UXShape>
+ CreateLine(params):List<UXShape>
+ TranslateShape(params):List<UXShape>
+ RotateShape(params):List<UXShape>
+ ChangeLineWidth(float, Uid):List<UXShape>
+ ChangeShapeFill (Color, Uid):List<UXShape>
+ ChangeStrokeColor(Color, Uid):List<UXShape>
+ ResizeShape(Cord, Cord, Uid):List<UXShape>
+ ChangeHeight():List<UXShape>
+ ChangeWidth():List<UXShape>
+ Undo():List<UXShape>
+ Redo():List<UXShape>
+ getUsername(Uid):String
+ switchState(State):Boolean
implements
BoardOperationsHandler
BoardOperationsState
compose
UserLevelHandler
inherits
inherits
ActiveBoardOperationsHandler
InactiveBoardOperationsHandler
Compose
Aggregates
Aggregates
Aggregates
UXShape
BoardShape
Aggregates
BoardServerShape
Aggregates
Convertor
ClientBoardStateManager

Fig 1: UML Diagram for Shape Operations

7.3 Features provided

a. Supported shapes
   i. Ellipse - The height, width of the ellipse.
   ii. Rectangle - Diagonally opposite points
   iii. Polyline - A set of points, representative of freehand drawing. The points progressively are stored, while real-time object creation.
   iv. Line - Using start and endpoints.

b. Supported operations on shapes
   i. Translation
   ii. Rotation
   iii. Resizing
   iv. Changing Height/Width
   v. Changing color, stroke width, shape fill color

c. Other operations:
   i. Undo/Redo
   ii. Finding Username corresponding to shape provided
   iii. Switching state between Active and Inactive modes.
   iv. Freezing and Unfreezing the Whiteboard.

7.4 Design Choices:

7.4.1 Internal Storage of Objects

- The internal representation in the server is a **BoardShape** object which tries to conceal what shape it actually is for instance if it is a circle/square etc. This is done since the server and the Client-Side StateManagement are not concerned about what actually the object is. This **BoardShape** class consists of all params to completely define a shape, but there might be some properties that are useful for some Shapes while not for others for instance array of points, which is only useful in the case of the polyline, hence in other cases, it is initialized to a NULL value.

- **BoardShape** is different from the objects sent to the UX for rendering i.e **UXShape**. The Object sent to the server is a serializable version of the class named **BoardServerShape**.
- Systems.Windows.Shapes class is used both at the UX and the Whiteboard. Reason: Ease of rendering at UX side, gives base ground. for the addition of new innovative functions to the whiteboard.
- A deep copy is always sent to the Client and State management unless explicit changes in the same objects are demanded.

7.4.2 The output provided to the UX module:

- From the perspective of the UX team, only 2 operations exist: DELETE, CREATE. Hence, each operation/creation either on state or shape is modeled into a series of DELETE and CREATE operations. Hence, the output to the UX is sent as an object of the class UXShape, which contains the operation, System.windows.shape, its x and y coordinates, and the rotational angle at the least.

- UXShape Class would also contain constructors to obtain UXShape objects directly from other Object types like the BoardShape object.

7.4.3 Toggle between Active and Inactive States:

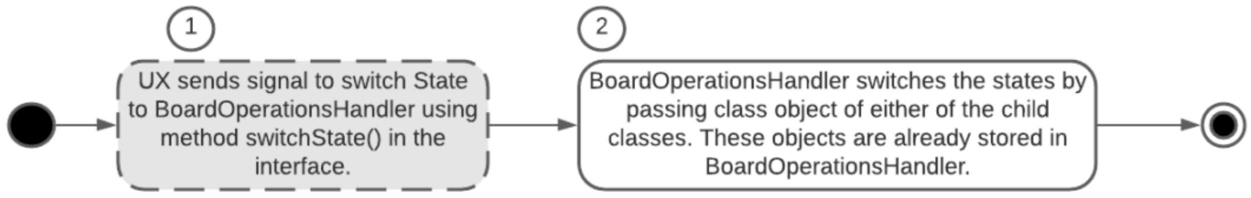

1
UX sends signal to switch State to BoardOperationsHandler using method switchState() in the interface.
2
BoardOperationsHandler switches the states by passing class object of either of the child classes. These objects are already stored in BoardOperationsHandler.

Fig 2: 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 SwitchState() method, the currentState object of class BoardOperationsState is toggled.

7.4.4 Real-Time Output UX of the Object

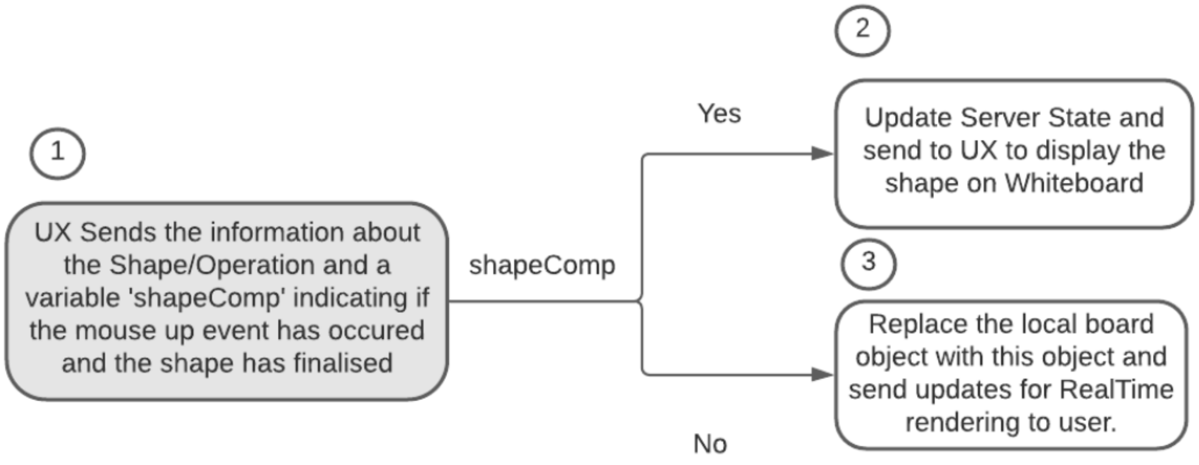

1
UX Sends the information about the Shape/Operation and a variable 'shapeComp' indicating if the mouse up event has occured and the shape has finalised
shapeComp
Yes
No
2
Update Server State and send to UX to display the shape on Whiteboard
3
Replace the local board object with this object and send updates for RealTime rendering to user.

Fig 3: Activity diagram for Real-time Output to UX.

BoardOperationsHandler receives operations on shapes or states related to the whiteboard. The variable shapeComp (1) 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. A series of operations are then sent to the UX module in the form of **List<UXShape>** to render on the whiteboard (2,3). These series of operations are performed using a temporary **BoardShape** object stored in the class **ActiveBoardOperationsHandler**. This keeps the track of the Shape just drawn, which needs to be deleted in the next call of the function.

7.4.5 Creation/ Deletion/ Modification of Shapes

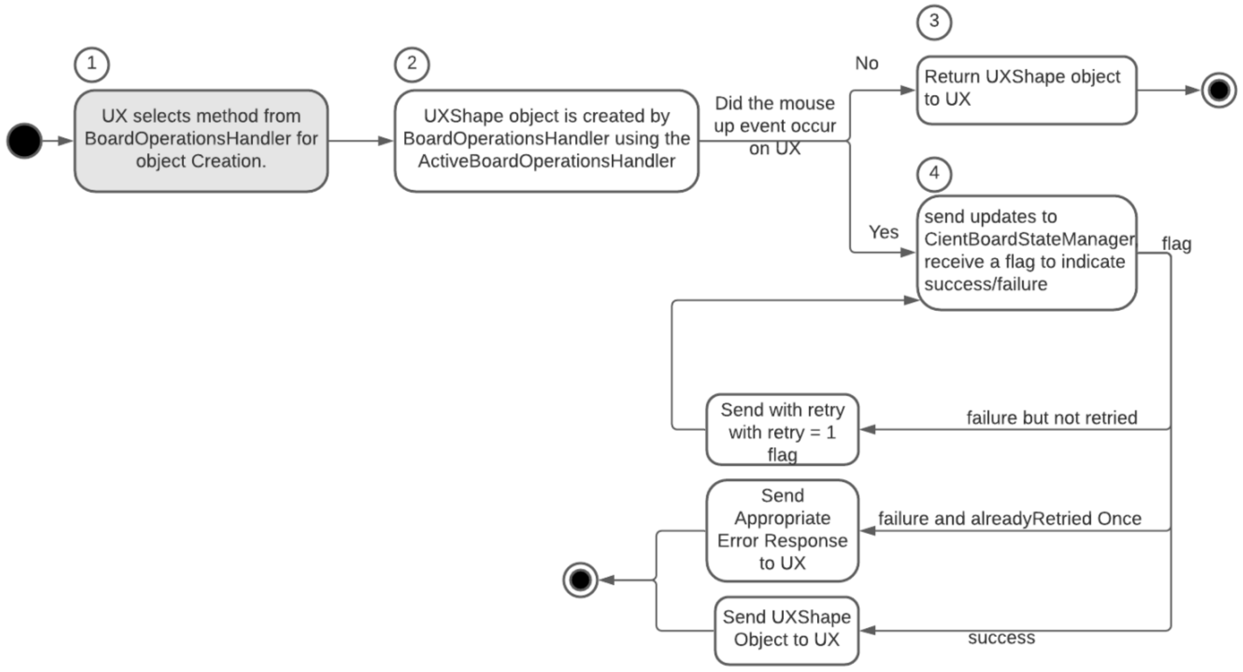

1
UX selects method from BoardOperationsHandler for object Creation.
2
UXShape object is created by BoardOperationsHandler using the ActiveBoardOperationsHandler
Did the mouse up event occur on UX
No
3
Return UXShape object to UX
Yes
4
send updates to CientBoardStateManager receive a flag to indicate success/failure
flag
Send with retry with retry = 1 flag
failure but not retried
Send Appropriate Error Response to UX
failure and alreadyRetried Once
Send UXShape Object to UX
success

Fig 4: Activity diagram for creation/state operation/shape operation

- The above Activity diagram assumed 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 and shape modification, the shape-specific attributes are calculated on the basis of the start-end coordinates, a whiteboard object is created. The previously-stored object, say prevBoardShape, is fetched(stored in ActiveBoardOperationsHandler for real-time rendering), coupled with DELETE Operation is sent, along with a new object UXShape with CREATE tag. If shapeComp is true, then updates are sent to the server. prevBoardShape is updated.
    - For the Undo-Redo operation, the ClientBoardStateManager itself performs Stack Updates, and return the list<UXShape> to the Handler, which in turns return it to the UX module.
- If there is an error in sending to UX, Handler tries again, else it quits with an appropriate error message.

Convertor Class:

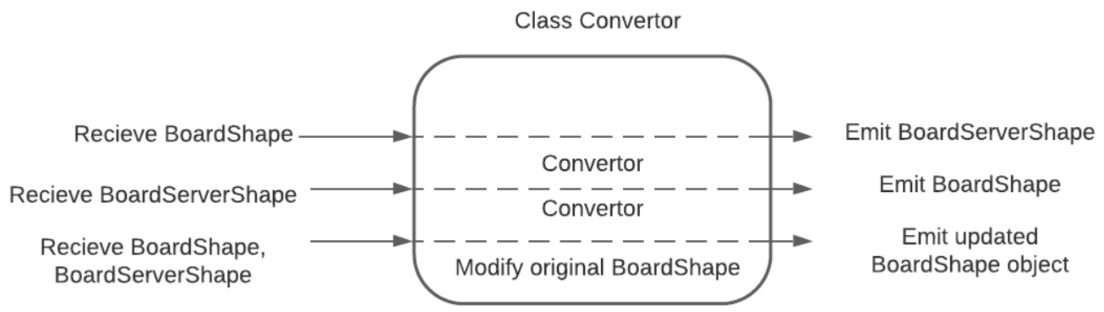


Fig 5: Convertor Class

- This class contains a map for conversion of internal params of class BoardShape which contains a non-serializable object System.Windows.Shapes to BoardServerObject and vice-versa.
- It also contains a function to modify an existing BoardShape using BoardServerObject.

```
Interface IConvertor{
        BoardShape ToBoardShape(BoardServerShape);
        BoardServerShape ToBoardServerShape(BoardShape);
        BoardShape ModifyOldBoardShape(BoardServerShape, BoardShape);
}
```

7.5 Challenges:

1. The System.Windows.Shapes is a UI element that easies rendering on the UX side. But this object only contained useful methods, and not directly useful information, moreover, this class is not serializable. Hence, additional classes had to be implemented to overcome these issues. Using this also brings UI-related elements in the state management part since the state should not use UI elements.

8. State Manager submodule

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:

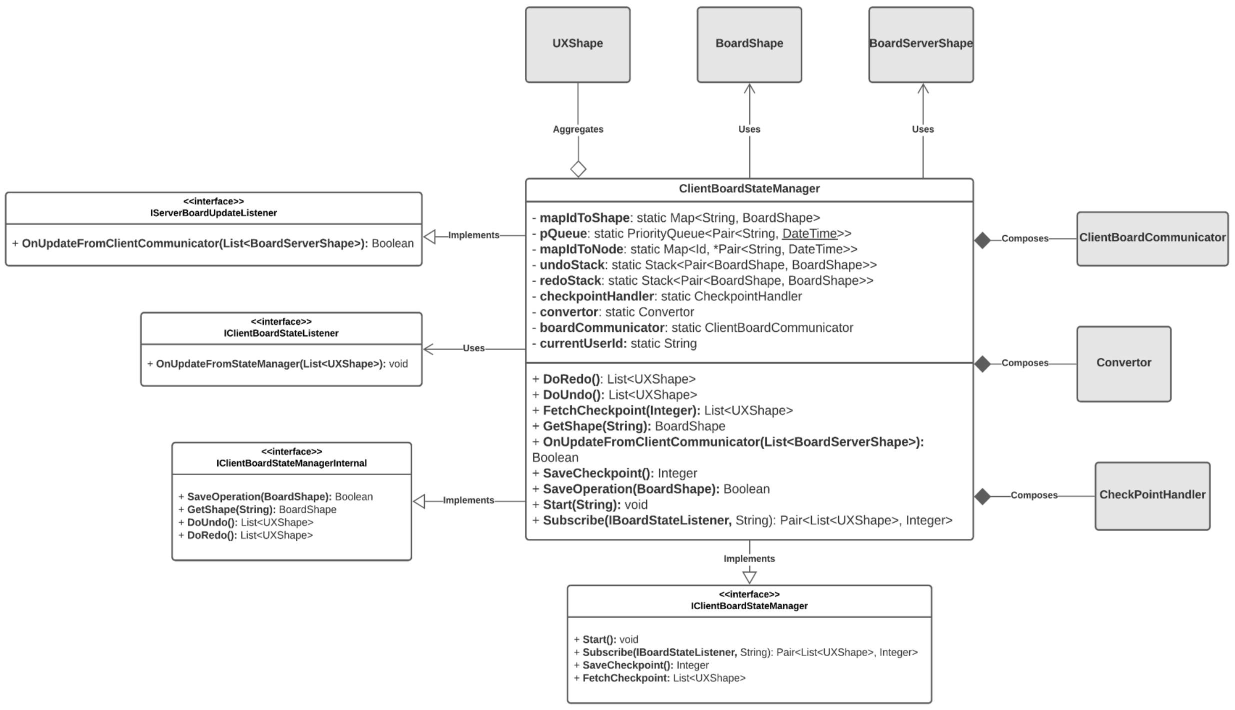

UXShape
BoardShape
BoardServerShape
Aggregates
Uses
Uses
ClientBoardStateManager
- mapIdToShape: static Map<String, BoardShape>
- pQueue: static PriorityQueue<Pair<String, DateTime>>
- mapIdToNode: static Map<Id, *Pair<String, DateTime>>
- undoStack: static Stack<Pair<BoardShape, BoardShape>>
- redoStack: static Stack<Pair<BoardShape, BoardShape>>
- checkpointHandler: static CheckpointHandler
- convertor: static Convertor
- boardCommunicator: static ClientBoardCommunicator
- currentUserId: static String
+ DoRedo(): List<UXShape>
+ DoUndo(): List<UXShape>
+ FetchCheckpoint(Integer): List<UXShape>
+ GetShape(String): BoardShape
+ OnUpdateFromClientCommunicator(List<BoardServerShape>): Boolean
+ SaveCheckpoint(): Integer
+ SaveOperation(BoardShape): Boolean
+ Start(String): void
+ Subscribe(IBoardStateListener, String): Pair<List<UXShape>, Integer>
<<interface>>
IServerBoardUpdateListener
+ OnUpdateFromClientCommunicator(List<BoardServerShape>): Boolean
Implements
<<interface>>
IClientBoardStateListener
+ OnUpdateFromStateManager(List<UXShape>): void
Uses
<<interface>>
IClientBoardStateManagerInternal
+ SaveOperation(BoardShape): Boolean
+ GetShape(String): BoardShape
+ DoUndo(): List<UXShape>
+ DoRedo(): List<UXShape>
Implements
Composes
ClientBoardCommunicator
Composes
Convertor
Composes
CheckPointHandler
Implements
<<interface>>
IClientBoardStateManager
+ Start(): void
+ Subscribe(IBoardStateListener, String): Pair<List<UXShape>, Integer>
+ SaveCheckpoint(): Integer
+ FetchCheckpoint: List<UXShape>

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 three 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 three 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 Pair of BoardShape ID and timestamp**: There will be a Priority queue (max heap) in which each node will contain two elements, BoardShape ID

(String) and timestamp (DateTime). 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 $O(klogn)$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 pointer to Priority Queue node**: 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 $O(logn)$ 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.

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 8th 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 $O(nlogn)$ 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.

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.

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.

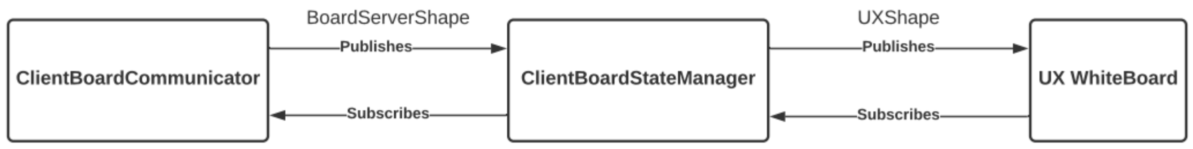


Figure 2. **Relationship between UX, State Manager and Communicator**: The communication objects are stated as well

Why this design choice?

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)

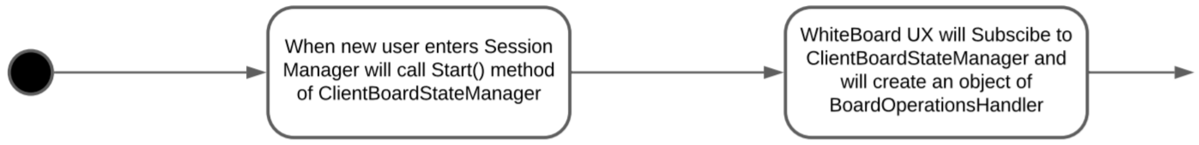

When new user enters Session Manager will call Start() method of ClientBoardStateManager
WhiteBoard UX will Subscibe to ClientBoardStateManager and will create an object of BoardOperationsHandler

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 class to subscribe to the Networking module.

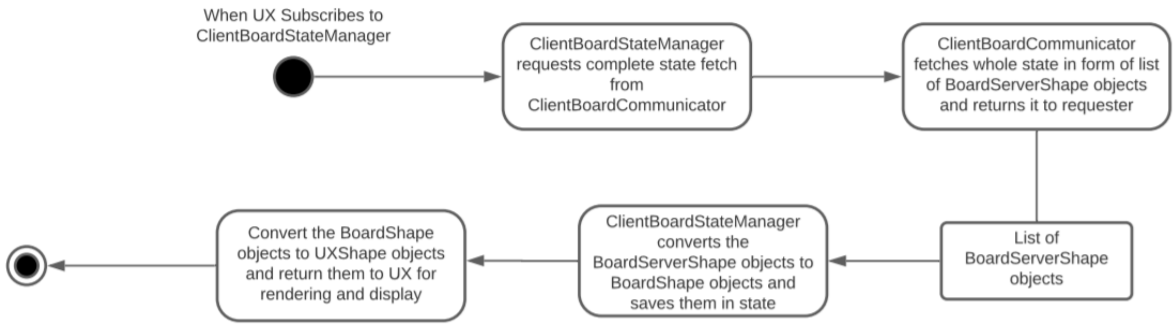
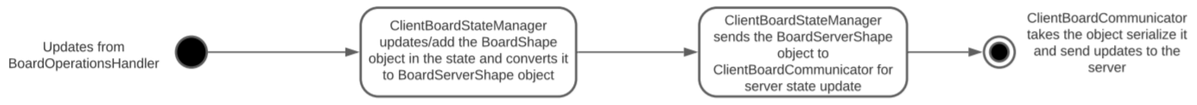

When UX Subscribes to ClientBoardStateManager
ClientBoardStateManager requests complete state fetch from ClientBoardCommunicator
ClientBoardCommunicator fetches whole state in form of list of BoardServerShape objects and returns it to requester
List of BoardServerShape objects
ClientBoardStateManager converts the BoardServerShape objects to BoardShape objects and saves them in state
Convert the BoardShape objects to UXShape objects and return them to UX for rendering and display

Figure 4. When UX Subscribes to ClientBoardStateManager

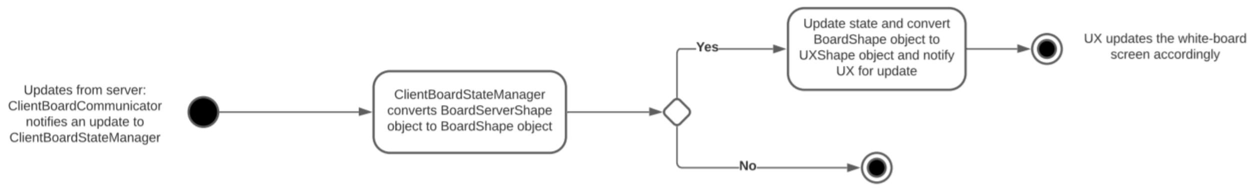
When UX subscribes to ClientBoardStateManager, the whole state which is saved at server is brought to this client via ClientBoardCommunicator which fetches the whole state. Then UX is provided with the sorted list of UXShape (sorted based on timestamp), so that the order of appearance of those shapes 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 option to switch to previous checkpoints as well.



Updates from BoardOperationsHandler
ClientBoardStateManager updates/add the BoardShape object in the state and converts it to BoardServerShape object
ClientBoardStateManager sends the BoardServerShape object to ClientBoardCommunicator for server state update
ClientBoardCommunicator takes the object serialize it and send updates to the server

Figure 5. On client-side operations

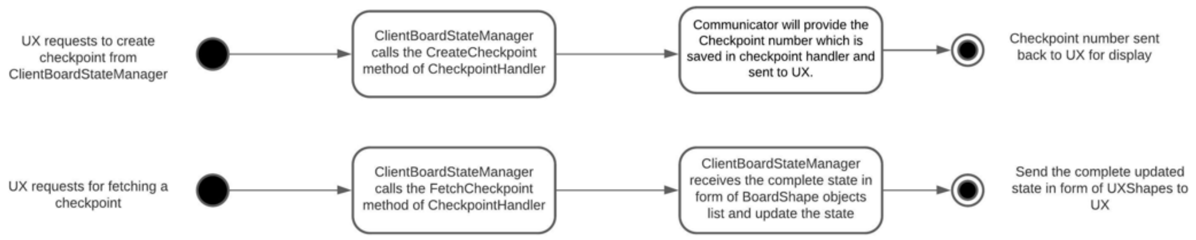
On any client-side operation, first the state of the client is updated, then the update is sent to ClientBoardCommunicator to send it to server from where the update is broadcasted to all clients and that update is saved in server's state as well.



Updates from server: ClientBoardCommunicator notifies an update to ClientBoardStateManager
ClientBoardStateManager converts BoardServerShape object to BoardShape object
Yes
Update state and convert BoardShape object to UXShape object and notify UX for update
UX updates the white-board screen accordingly
No

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.

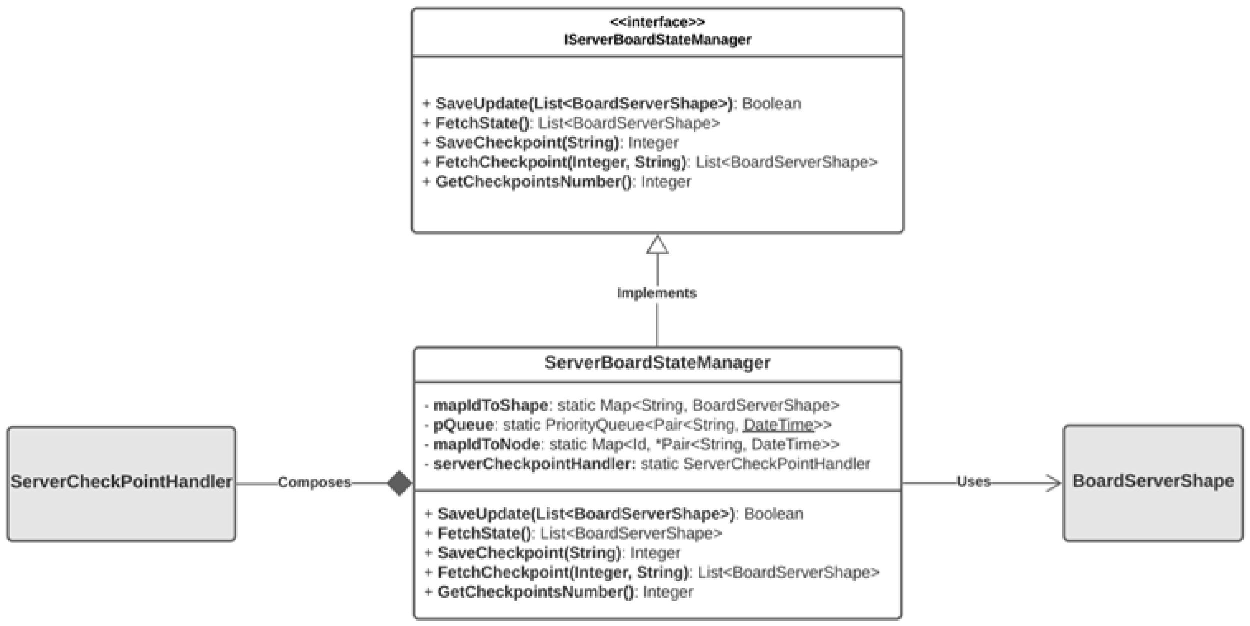


UX requests to create checkpoint from ClientBoardStateManager
ClientBoardStateManager calls the CreateCheckpoint method of CheckpointHandler
Communicator will provide the Checkpoint number which is saved in checkpoint handler and sent to UX.
Checkpoint number sent back to UX for display
UX requests for fetching a checkpoint
ClientBoardStateManager calls the FetchCheckpoint method of CheckpointHandler
ClientBoardStateManager receives the complete state in form of BoardShape objects list and update the state
Send the complete updated state in form of UXShapes to UX

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:



<<interface>>
IServerBoardStateManager
+ SaveUpdate(List<BoardServerShape>): Boolean
+ FetchState(): List<BoardServerShape>
+ SaveCheckpoint(String): Integer
+ FetchCheckpoint(Integer, String): List<BoardServerShape>
+ GetCheckpointsNumber(): Integer
Implements
ServerBoardStateManager
- mapIdToShape: static Map<String, BoardServerShape>
- pQueue: static PriorityQueue<Pair<String, DateTime>>
- mapIdToNode: static Map<Id, *Pair<String, DateTime>>
- serverCheckpointHandler: static ServerCheckPointHandler
+ SaveUpdate(List<BoardServerShape>): Boolean
+ FetchState(): List<BoardServerShape>
+ SaveCheckpoint(String): Integer
+ FetchCheckpoint(Integer, String): List<BoardServerShape>
+ GetCheckpointsNumber(): Integer
ServerCheckPointHandler
Composes
Uses
BoardServerShape

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. However, there is one major difference. Here we will be storing BoardServerShape unlike BoardShape objects which are stored at the client side. The reason is BoardShape consists of UIElements such as System.Windows.Shape objects which are not serializable and hence can't be used for network communication. Since, at the server we are not going to have a display, we can just store all the information about shape in the form of BoardServerShape objects. The state manager at client side will take care of all the conversions from BoardServerShape to BoardShape and vice-versa. Another subtle difference is that here there will be no undo-redo stacks since they are not required at server-side.

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

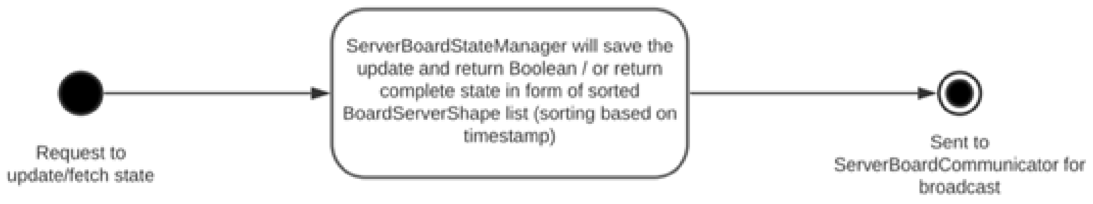

Request to update/fetch state
ServerBoardStateManager will save the update and return Boolean / or return complete state in form of sorted BoardServerShape list (sorting based on timestamp)
Sent to ServerBoardCommunicator for broadcast

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 sorted list (based on timestamp) of BoardServerShape and provide it to ServerBoardCommunicator.

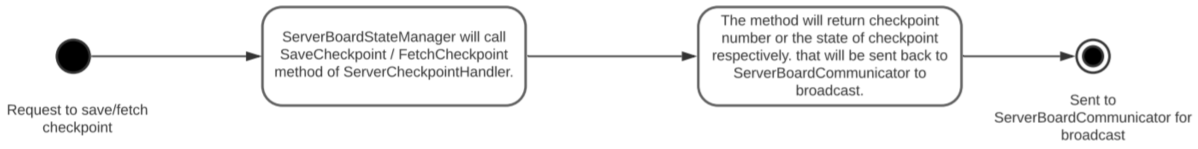

Request to save/fetch checkpoint
ServerBoardStateManager will call SaveCheckpoint / FetchCheckpoint method of ServerCheckpointHandler.
The method will return checkpoint number or the state of checkpoint respectively. that will be sent back to ServerBoardCommunicator to broadcast.
Sent to ServerBoardCommunicator for broadcast

Figure 10. Interactions with ServerCheckpointHandler

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

9. Checkpoint Handler submodule

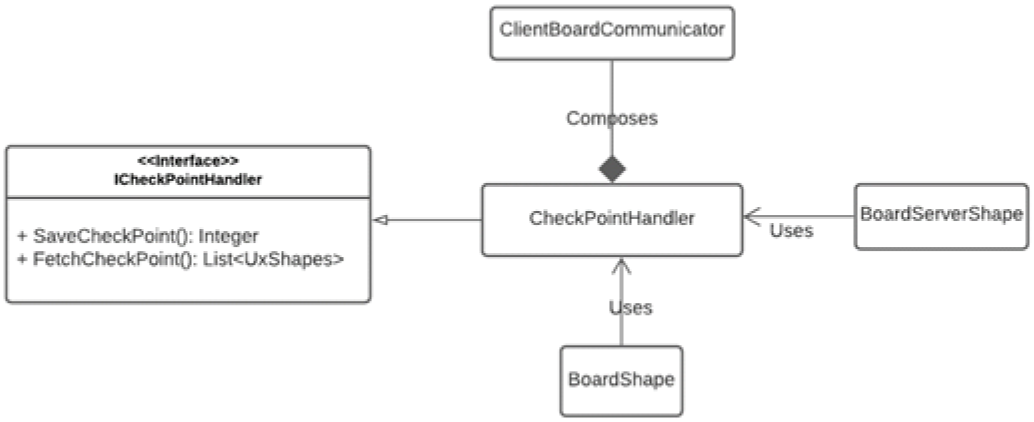
9.1 Objectives

- 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.

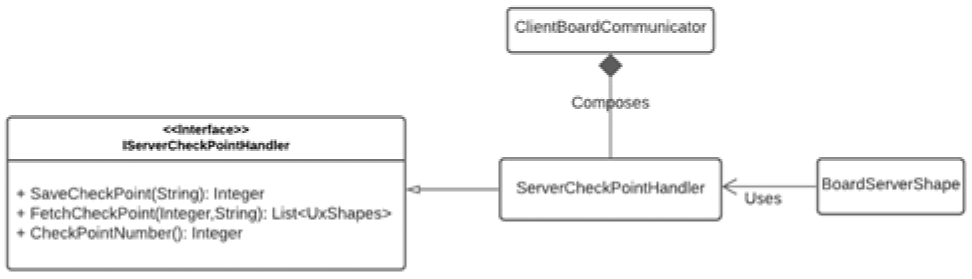
9.2 Design

Client Side:



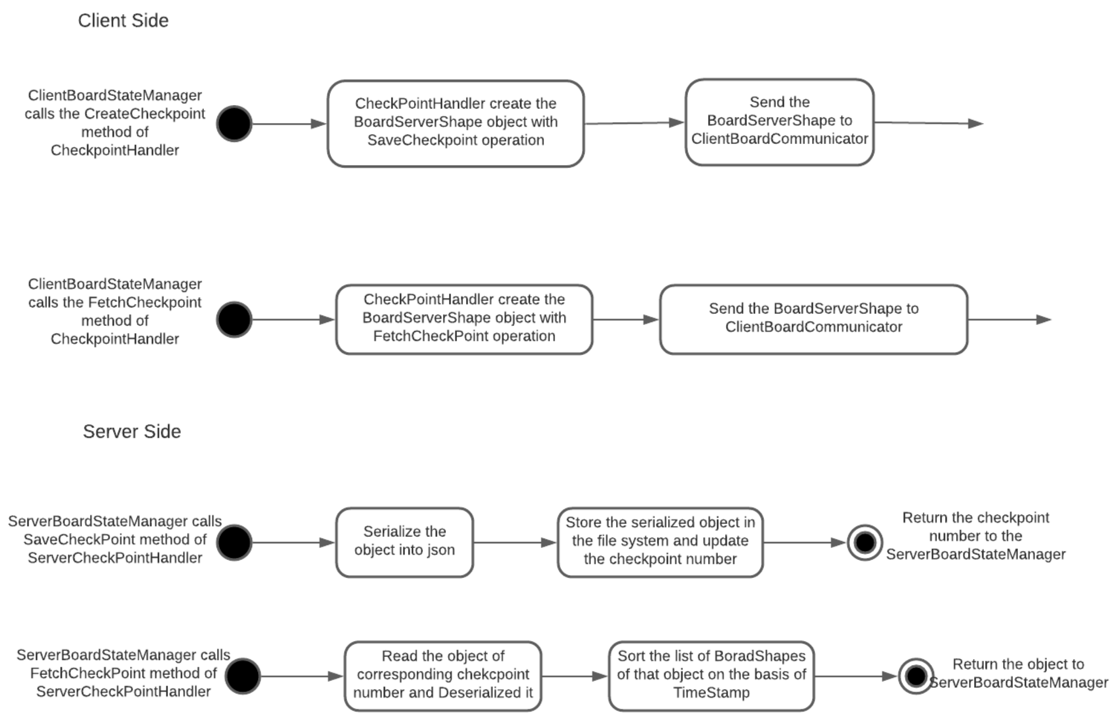
This class CheckPointHandler is composed by ClientBoardStateManager to do save and fetch checkpoint. CheckPointHandler creates BoardServerShapes object with SaveCheckpoint/ FetchCheckpoint operation and send it to ClientBoardCommunicator. ClientBoardCommunicator will do the required task and return the checkpointnumber/ board state to ClientBoardStateManager.

Server Side:


ClientBoardCommunicator
Composes
<<Interface>>
IServerCheckPointHandler
+ SaveCheckPoint(String): Integer
+ FetchCheckPoint(Integer,String): List<UxShapes>
+ CheckPointNumber(): Integer
ServerCheckPointHandler
Uses
BoardServerShape

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:


Client Side
ClientBoardStateManager calls the CreateCheckpoint method of CheckpointHandler
CheckPointHandler create the BoardServerShape object with SaveCheckpoint operation
Send the BoardServerShape to ClientBoardCommunicator
ClientBoardStateManager calls the FetchCheckpoint method of CheckpointHandler
CheckPointHandler create the BoardServerShape object with FetchCheckPoint operation
Send the BoardServerShape to ClientBoardCommunicator
Server Side
ServerBoardStateManager calls SaveCheckPoint method of ServerCheckPointHandler
Serialize the object into json
Store the serialized object in the file system and update the checkpoint number
Return the checkpoint number to the ServerBoardStateManager
ServerBoardStateManager calls FetchCheckPoint method of ServerCheckPointHandler
Read the object of corresponding chekcpoint number and Deserialized it
Sort the list of BoradShapes of that object on the basis of TimeStamp
Return the object to ServerBoardStateManager

SaveCheckPoint at client side:

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

FetchCheckPoint at client side:

- ClientBoardStateManger 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

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.

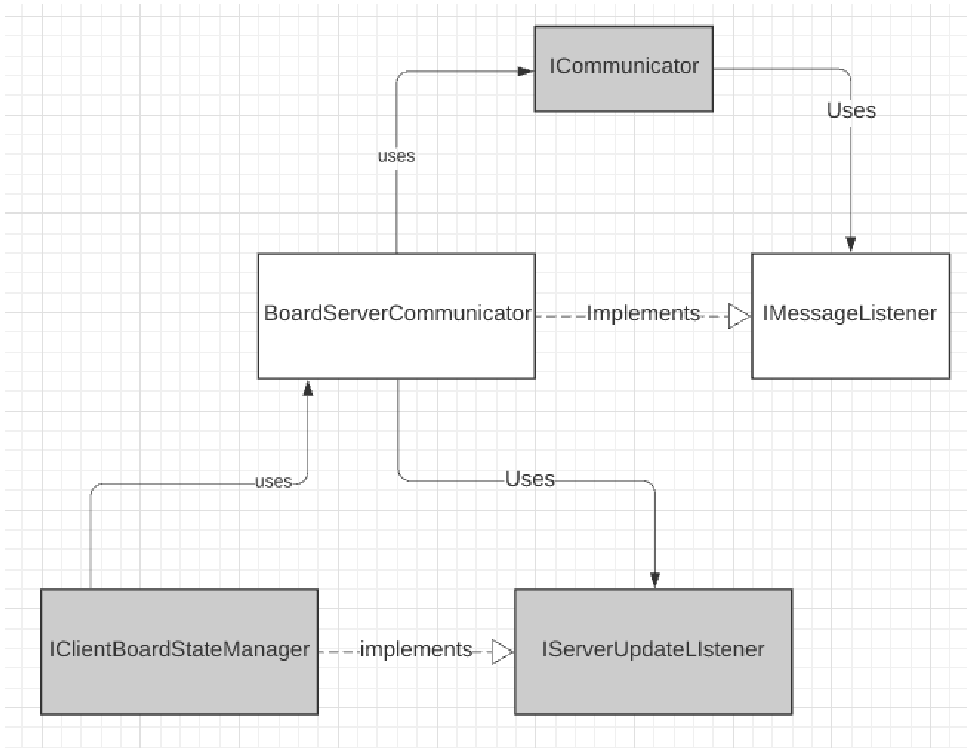
10.2 Objectives:

- 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.

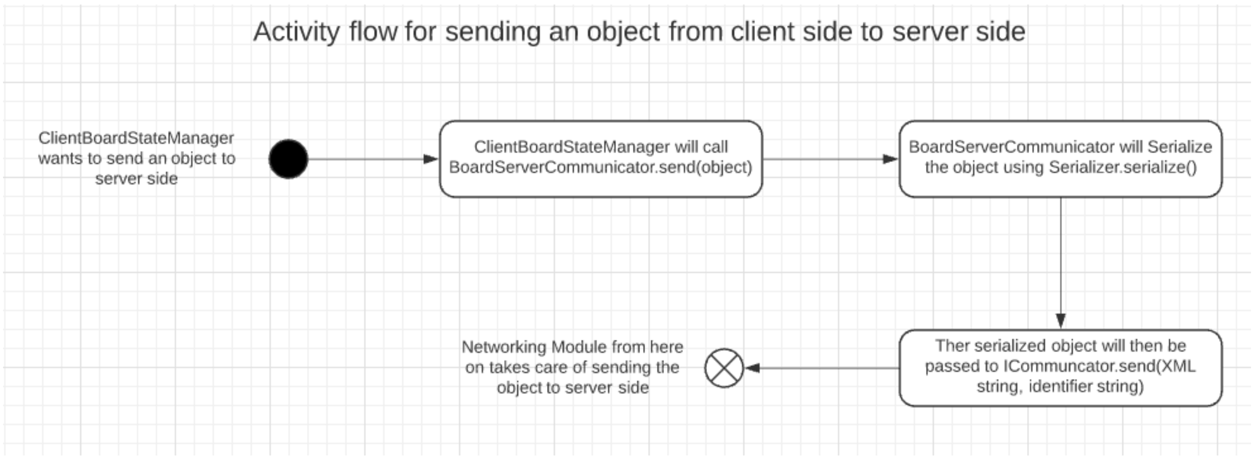
10.3 Design:

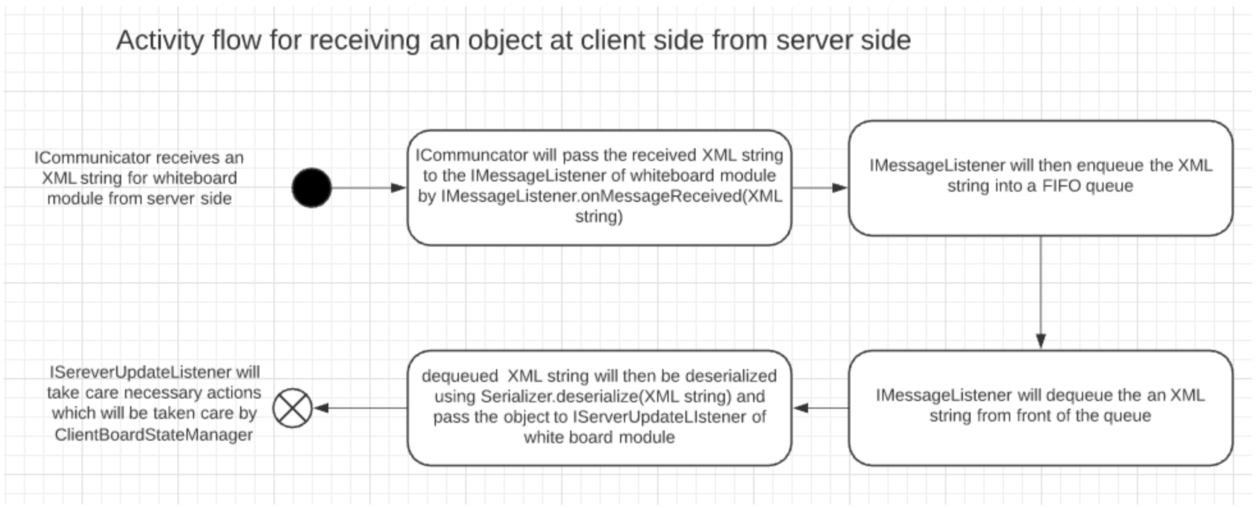
1. FIFO queue is chosen to store the objects that are received so that we give priority to the object that we received first
2. Publisher subscriber pattern is chosen to notify the client side state manager whenever we receive an object because it offers asynchronous message delivery. This pattern helps 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

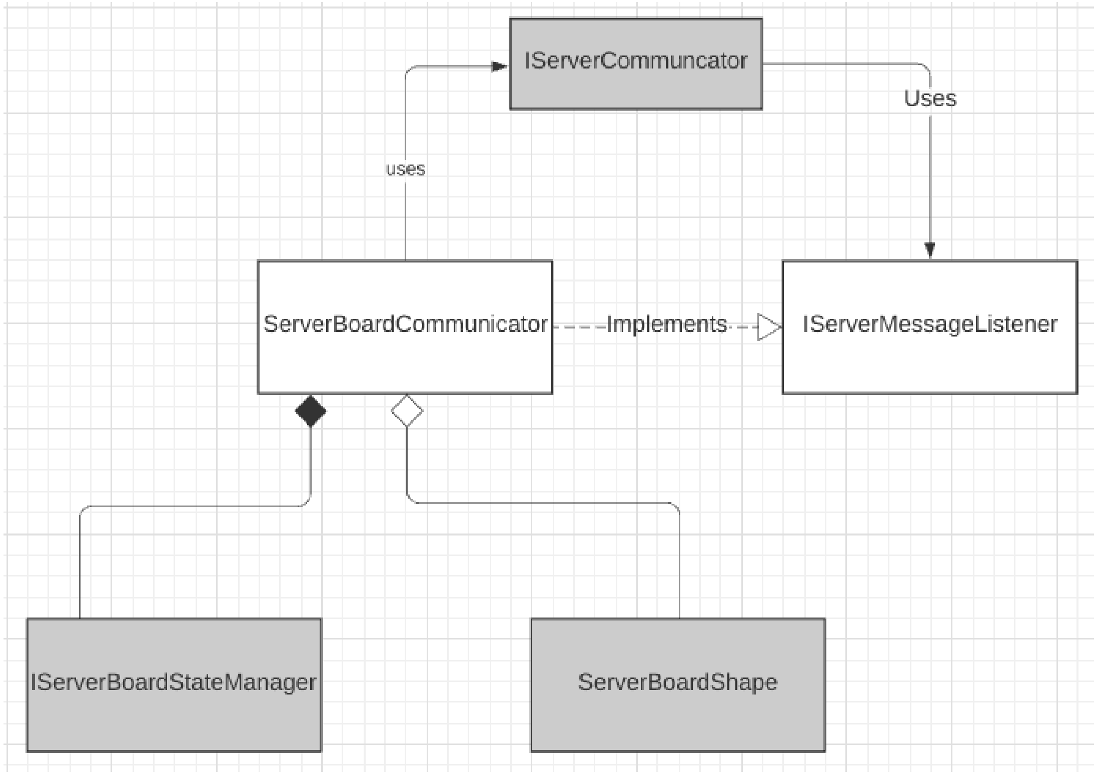


Class Diagram for BoardServerCommunicator at client side

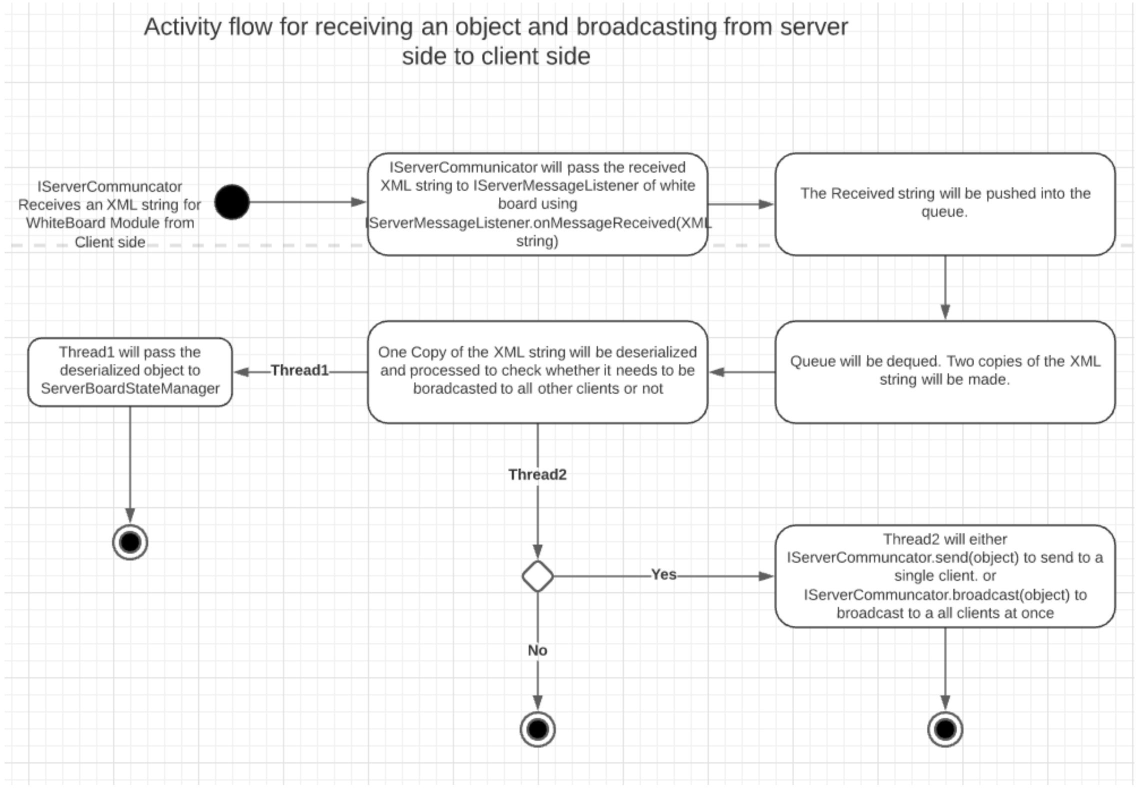

Activity flow for sending an object from client side to server side
ClientBoardStateManager wants to send an object to server side
ClientBoardStateManager will call BoardServerCommunicator.send(object)
BoardServerCommunicator will Serialize the object using Serializer.serialize()
Ther serialized object will then be passed to ICommuncator.send(XML string, identifier string)
Networking Module from here on takes care of sending the object to server side


Activity flow for receiving an object at client side from server side
ICommunicator receives an XML string for whiteboard module from server side
ICommuncator will pass the received XML string to the IMessageListener of whiteboard module by IMessageListener.onMessageReceived(XML string)
IMessageListener will then enqueue the XML string into a FIFO queue
IMessageListener will dequeue the an XML string from front of the queue
dequeued XML string will then be deserialized using Serializer.deserialize(XML string) and pass the object to IServerUpdateLIstener of white board module
ISereverUpdateListener will take care necessary actions which will be taken care by ClientBoardStateManager

- *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



Activity flow for receiving an object and broadcasting from server side to client side
IServerCommuncator Receives an XML string for WhiteBoard Module from Client side
IServerCommunicator will pass the received XML string to IServerMessageListener of white board using IServerMessageListener.onMessageReceived(XML string)
The Received string will be pushed into the queue.
Queue will be dequed. Two copies of the XML string will be made.
One Copy of the XML string will be deserialized and processed to check whether it needs to be boradcasted to all other clients or not
Thread1
Thread1 will pass the deserialized object to ServerBoardStateManager
Thread2
Yes
No
Thread2 will either IServerCommuncator.send(object) to send to a single client. or IServerCommuncator.broadcast(object) to broadcast to a all clients at once

- 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 *icommunicator.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

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

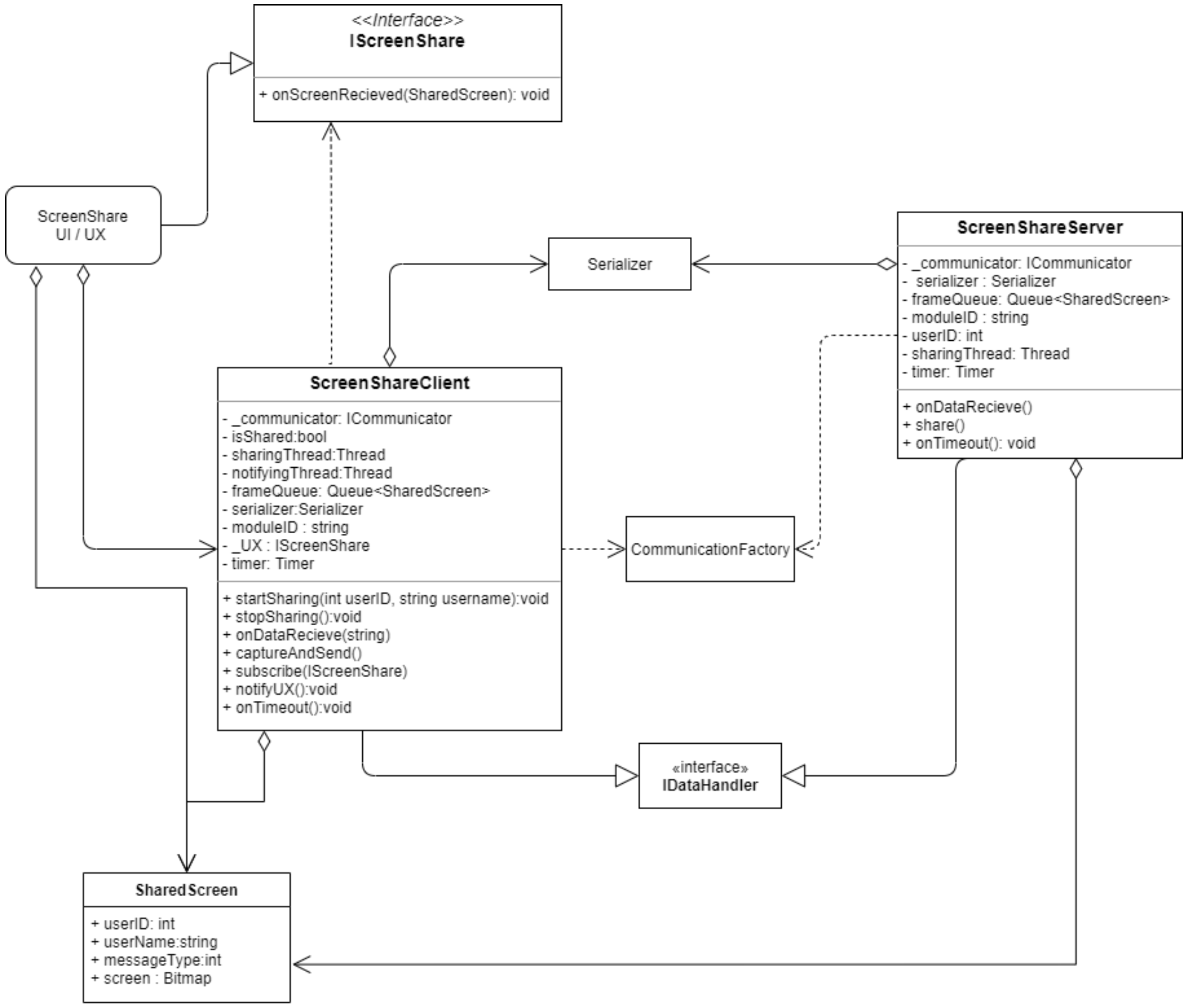
11.2 Objectives:

- **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’.

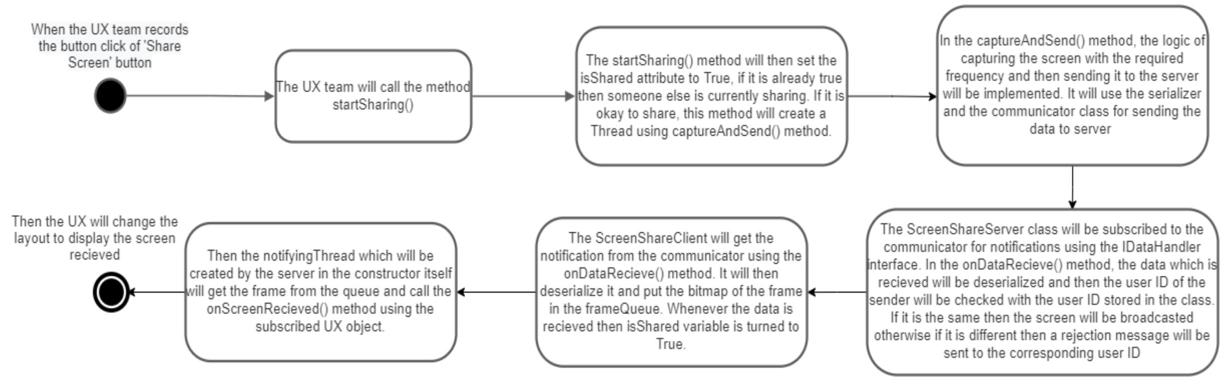
- **ScreenShareServer:**
  - The constructor of this class will create an instance of the communicator class using the ‘CommunicationFactory’, it will also create an instance of the serializer.
  - It will maintain a ‘frameQueue’ to organize the incoming frames. This ‘frameQueue’ will be filled by the ‘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


When the UX team records the button click of 'Share Screen' button
The UX team will call the method startSharing()
The startSharing() method will then set the isShared attribute to True, if it is already true then someone else is currently sharing. If it is okay to share, this method will create a Thread using captureAndSend() method.
In the captureAndSend() method, the logic of capturing the screen with the required frequency and then sending it to the server will be implemented. It will use the serializer and the communicator class for sending the data to server
The ScreenShareServer class will be subscribed to the communicator for notifications using the IDataHandler interface. In the onDataRecieve() method, the data which is recieved will be deserialized and then the user ID of the sender will be checked with the user ID stored in the class. If it is the same then the screen will be broadcasted otherwise if it is different then a rejection message will be sent to the corresponding user ID
The ScreenShareClient will get the notification from the communicator using the onDataRecieve() method. It will then deserialize it and put the bitmap of the frame in the frameQueue. Whenever the data is recieved then isShared variable is turned to True.
Then the notifyingThread which will be created by the server in the constructor itself will get the frame from the queue and call the onScreenRecieved() method using the subscribed UX object.
Then the UX will change the layout to display the screen recieved

2. Stop Sharing the screen

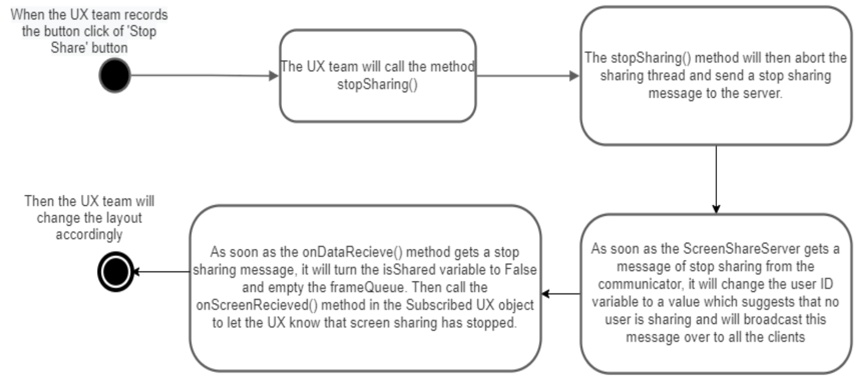
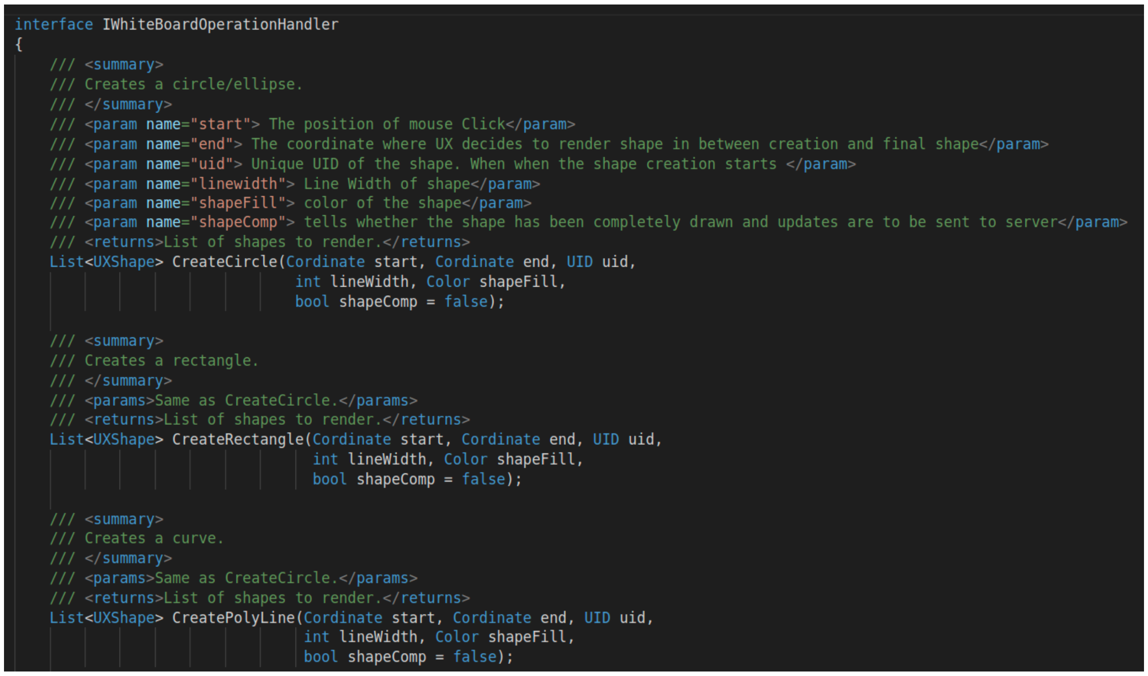

When the UX team records the button click of 'Stop Share' button
The UX team will call the method stopSharing()
The stopSharing() method will then abort the sharing thread and send a stop sharing message to the server.
As soon as the ScreenShareServer gets a message of stop sharing from the communicator, it will change the user ID variable to a value which suggests that no user is sharing and will broadcast this message over to all the clients
As soon as the onDataRecieve() method gets a stop sharing message, it will turn the isShared variable to False and empty the frameQueue. Then call the onScreenRecieved() method in the Subscribed UX object to let the UX know that screen sharing has stopped.
Then the UX team will change the layout accordingly

Figure: Activity diagram denoting the workflow

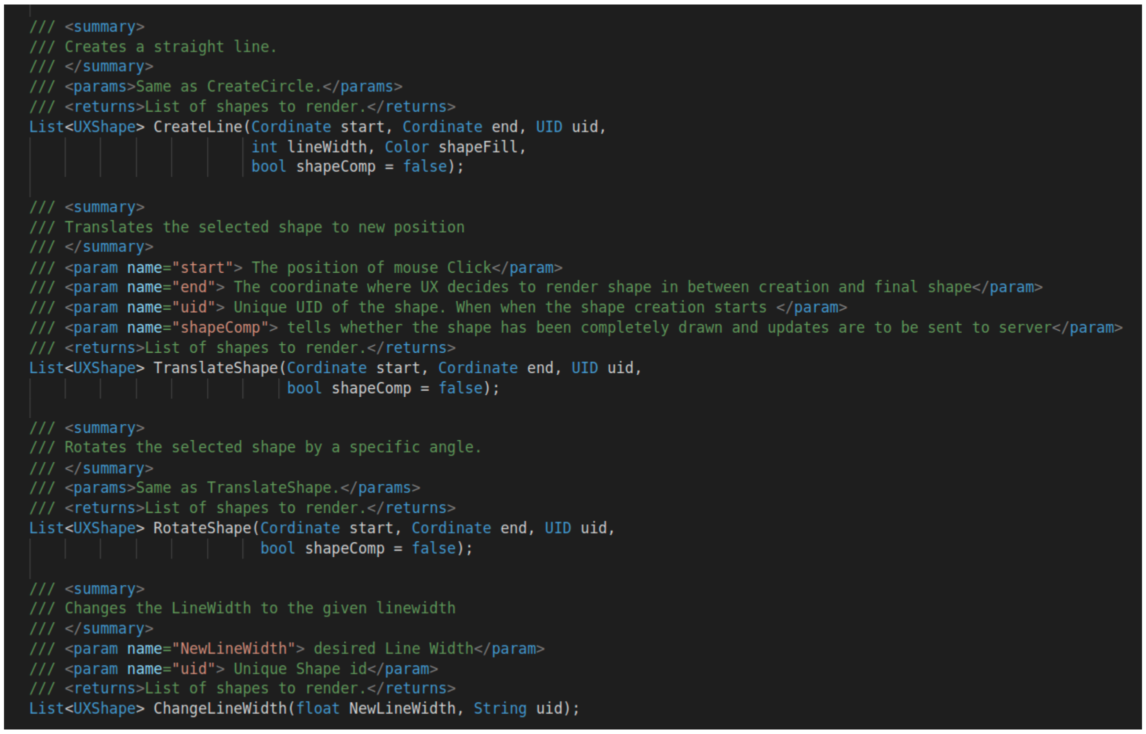
12. Interfaces

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

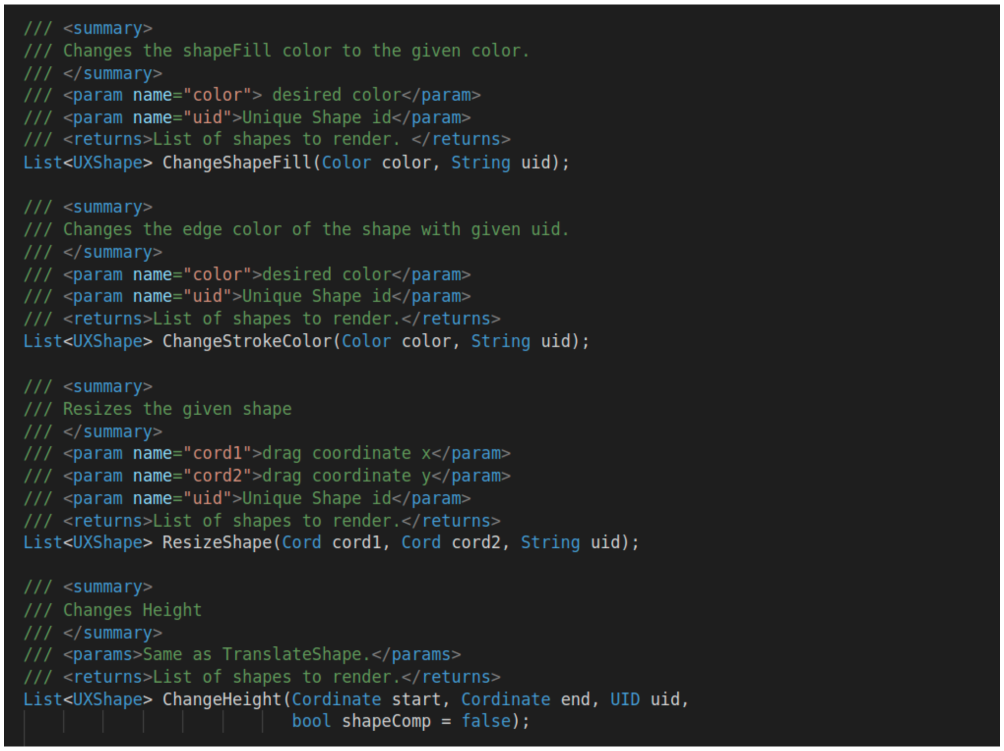
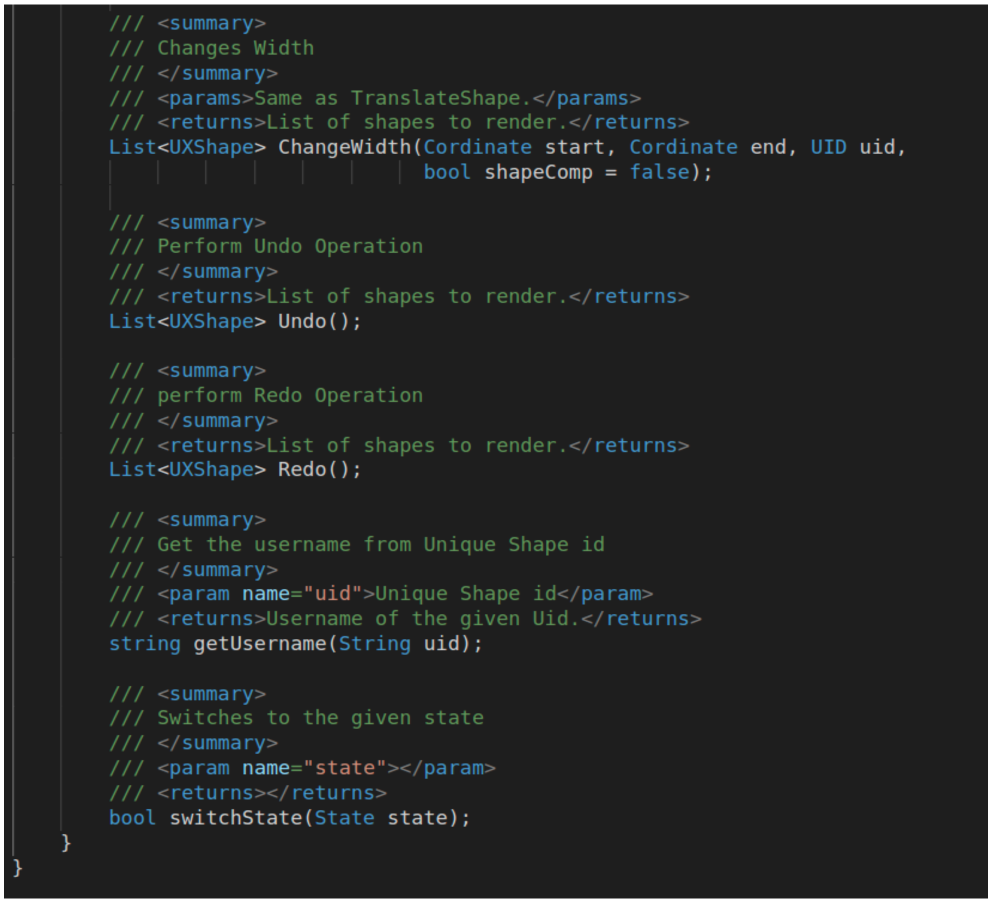
1. **IWhiteBoardOperationHandler**



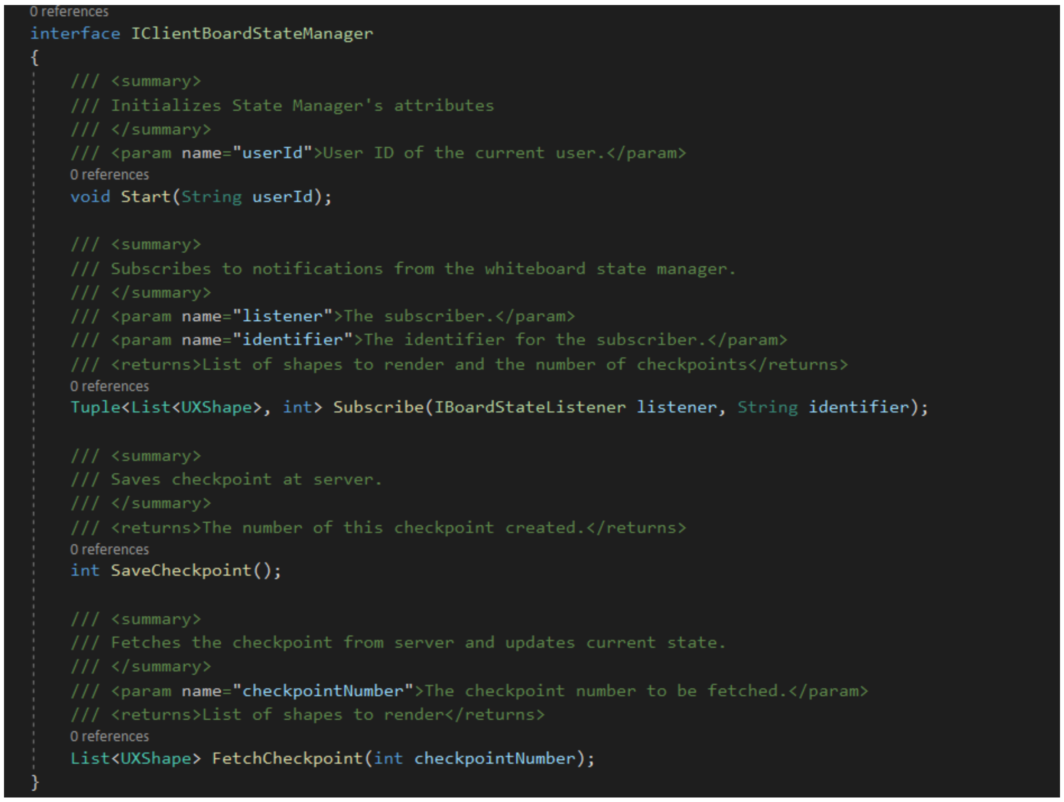
```
interface IWhiteBoardOperationHandler
{
    /// <summary>
    /// Creates a circle/ellipse.
    /// </summary>
    /// <param name="start"> The position of mouse Click</param>
    /// <param name="end"> The coordinate where UX decides to render shape in between creation and final shape</param>
    /// <param name="uid"> Unique UID of the shape. When when the shape creation starts </param>
    /// <param name="linewidth"> Line Width of shape</param>
    /// <param name="shapeFill"> color of the shape</param>
    /// <param name="shapeComp"> tells whether the shape has been completely drawn and updates are to be sent to server</param>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> CreateCircle(Cordinate start, Cordinate end, UID uid,
                               int lineWidth, Color shapeFill,
                               bool shapeComp = false);

    /// <summary>
    /// Creates a rectangle.
    /// </summary>
    /// <params>Same as CreateCircle.</params>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> CreateRectangle(Cordinate start, Cordinate end, UID uid,
                                  int lineWidth, Color shapeFill,
                                  bool shapeComp = false);

    /// <summary>
    /// Creates a curve.
    /// </summary>
    /// <params>Same as CreateCircle.</params>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> CreatePolyLine(Cordinate start, Cordinate end, UID uid,
                                 int lineWidth, Color shapeFill,
                                 bool shapeComp = false);
```



```
    /// <summary>
    /// Creates a straight line.
    /// </summary>
    /// <params>Same as CreateCircle.</params>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> CreateLine(Cordinate start, Cordinate end, UID uid,
                             int lineWidth, Color shapeFill,
                             bool shapeComp = false);

    /// <summary>
    /// Translates the selected shape to new position
    /// </summary>
    /// <param name="start"> The position of mouse Click</param>
    /// <param name="end"> The coordinate where UX decides to render shape in between creation and final shape</param>
    /// <param name="uid"> Unique UID of the shape. When when the shape creation starts </param>
    /// <param name="shapeComp"> tells whether the shape has been completely drawn and updates are to be sent to server</param>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> TranslateShape(Cordinate start, Cordinate end, UID uid,
                                 bool shapeComp = false);

    /// <summary>
    /// Rotates the selected shape by a specific angle.
    /// </summary>
    /// <params>Same as TranslateShape.</params>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> RotateShape(Cordinate start, Cordinate end, UID uid,
                              bool shapeComp = false);

    /// <summary>
    /// Changes the LineWidth to the given linewidth
    /// </summary>
    /// <param name="NewLineWidth"> desired Line Width</param>
    /// <param name="uid"> Unique Shape id</param>
    /// <returns>List of shapes to render.</returns>
    List<UXShape> ChangeLineWidth(float NewLineWidth, String uid);
```


```
/// <summary>
/// Changes the shapeFill color to the given color.
/// </summary>
/// <param name="color"> desired color</param>
/// <param name="uid">Unique Shape id</param>
/// <returns>List of shapes to render. </returns>
List<UXShape> ChangeShapeFill(Color color, String uid);

/// <summary>
/// Changes the edge color of the shape with given uid.
/// </summary>
/// <param name="color">desired color</param>
/// <param name="uid">Unique Shape id</param>
/// <returns>List of shapes to render.</returns>
List<UXShape> ChangeStrokeColor(Color color, String uid);

/// <summary>
/// Resizes the given shape
/// </summary>
/// <param name="cord1">drag coordinate x</param>
/// <param name="cord2">drag coordinate y</param>
/// <param name="uid">Unique Shape id</param>
/// <returns>List of shapes to render.</returns>
List<UXShape> ResizeShape(Cord cord1, Cord cord2, String uid);

/// <summary>
/// Changes Height
/// </summary>
/// <params>Same as TranslateShape.</params>
/// <returns>List of shapes to render.</returns>
List<UXShape> ChangeHeight(Cordinate start, Cordinate end, UID uid,
                           bool shapeComp = false);
```

```
        /// <summary>
        /// Changes Width
        /// </summary>
        /// <params>Same as TranslateShape.</params>
        /// <returns>List of shapes to render.</returns>
        List<UXShape> ChangeWidth(Cordinate start, Cordinate end, UID uid,
                                  bool shapeComp = false);

        /// <summary>
        /// Perform Undo Operation
        /// </summary>
        /// <returns>List of shapes to render.</returns>
        List<UXShape> Undo();

        /// <summary>
        /// perform Redo Operation
        /// </summary>
        /// <returns>List of shapes to render.</returns>
        List<UXShape> Redo();

        /// <summary>
        /// Get the username from Unique Shape id
        /// </summary>
        /// <param name="uid">Unique Shape id</param>
        /// <returns>Username of the given Uid.</returns>
        string getUsername(String uid);

        /// <summary>
        /// Switches to the given state
        /// </summary>
        /// <param name="state"></param>
        /// <returns></returns>
        bool switchState(State state);
    }
}
```

2. IClientBoardStateManager



```
0 references
interface IClientBoardStateManager
{
    /// <summary>
    /// Initializes State Manager's attributes
    /// </summary>
    /// <param name="userId">User ID of the current user.</param>
    0 references
    void Start(String userId);

    /// <summary>
    /// Subscribes to notifications from the whiteboard state manager.
    /// </summary>
    /// <param name="listener">The subscriber.</param>
    /// <param name="identifier">The identifier for the subscriber.</param>
    /// <returns>List of shapes to render and the number of checkpoints</returns>
    0 references
    Tuple<List<UXShape>, int> Subscribe(IBoardStateListener listener, String identifier);

    /// <summary>
    /// Saves checkpoint at server.
    /// </summary>
    /// <returns>The number of this checkpoint created.</returns>
    0 references
    int SaveCheckpoint();

    /// <summary>
    /// Fetches the checkpoint from server and updates current state.
    /// </summary>
    /// <param name="checkpointNumber">The checkpoint number to be fetched.</param>
    /// <returns>List of shapes to render</returns>
    0 references
    List<UXShape> FetchCheckpoint(int checkpointNumber);
}
```

3. IMessageListener and IServerMessageListener

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

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.