Shared Whiteboard

Yuqing Chang
Student Number: 1044862
yuqchang@student.unimelb.edu.au

1 Introduction

This project designs and implements a shared whiteboard, allowing multiple users to draw simultaneously on a canvas. Users can use a pen, eraser, text tool and draw basic shapes, such as lines, circles, ovals, rectangles. The communication between server and users is based on remote interfaces.

This project is based on Java RMI Remote interface; each function in it should throw RemoteException. Any object that is remote must implement this interface. Only those methods specified in a ”remote interface” are available remotely. Once the first user opens the GUI, he will be automatically connected to the RMI agency and become the manager.

In this report, section 2 introduces process of using the system and main basic functionalities. Section 3 covers overall class design and Creativity elements. It also contains advanced features, such as chat room, kick out a particular user, manager's right to open, save, save as, new and close the whiteboard. Section 3 also contains UML class diagram and an interaction diagram, and descriptions about each class. Section 4 covers critical analysis about this project. I have reflected on the shortcomings of the design idea and have proposed ways to improve it. Section 5 is conclusion of what has been learnt in this project.

I use TeXstudio to do LateX format. This report's format is in 10pt, one-columns, left margin and right margin are 1.9 cm.

2 Components

At the beginning of the project, we should run a server to create an instance of the *IRemoteBoard* class and publish the remote object's stub in the registry under the name ”Whiteboard”. Then the first user runs *CreateWhiteBoard* to create a whiteboard and becomes the manager of the board. Other users can run *joinWhiteBoard* to ask for permission to enter the board. Because the “username” is unique, they will be checked if their usernames currently exist on the whiteboard. If not, a pop-up window will appear on the manager side to ask if the user can enter the whiteboard. Once manager gives approvals, all users can enter and draw concurrently on the board.

During the system, the manager can kick out any user at any time by inputting their username. And user side will appear a pop-up window to inform the user that the manager has kicked out him. At the same time, all active users can receive a message that says “XXX is kicked out by the manager” in chat room. Besides, the manager has the right to open, save, save as, new and close the whiteboard. Once the manager exits the canvas, all users will be forced to withdraw.

Each user's name and identity will be shown at the top left corner of GUI.

The main functionalities of whiteboard are the followings.

2.1 Draw and Text

Whiteboard interface has tool buttons, which contains "Line", "Circle", "Rectangle", "Oval", "Pen", "Eraser", "Text". Users can draw lines, circles, ovals, rectangles in the whiteboard, and use pen and eraser to paint freely. They also can insert a text anywhere on the drawing board. I implement these drawing functions based on Java2D drawing package to draw a certain shape. And I implement $MouseListene$, $MouseMotionListener$ interfaces to monitor mouse movement and click or drag position coordinates.

2.2 Choose a color to draw

User can choose their favorite color to draw the above shapes. The default color at the beginning is black. A simple color palette is on the top of canvas and contains standard colors, such as blue, yellow, pink, green and red. Besides, the user can click on "More colors" button to open a complete palette, which contains more than 128 colors.(Figure 1)

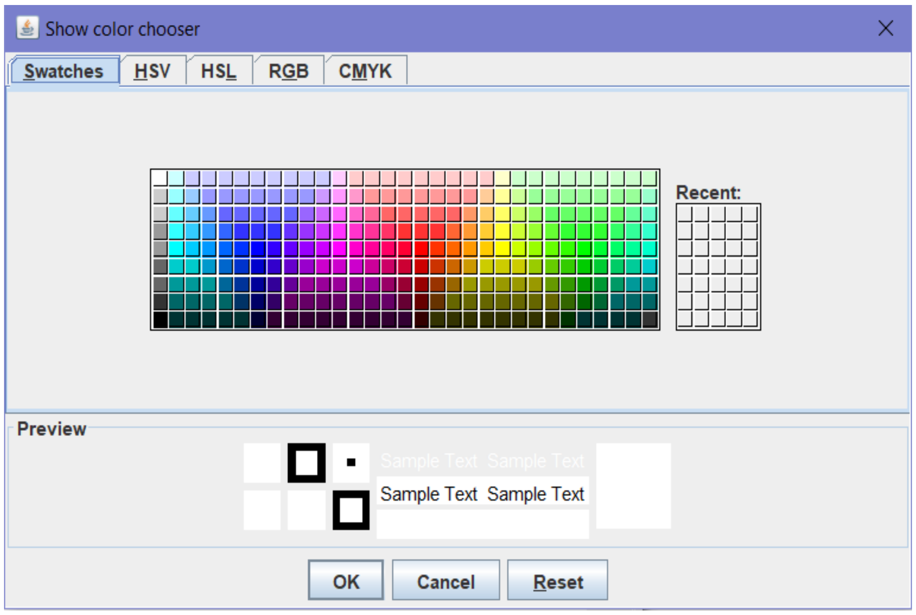


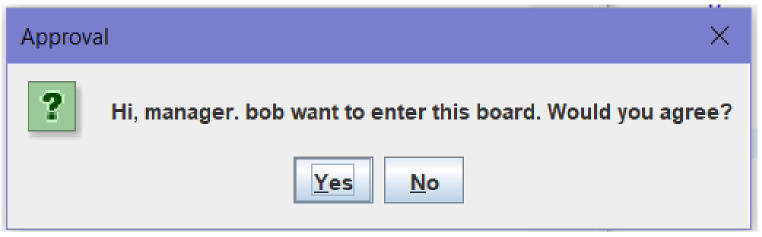
Figure 1: Color Chooser

2.3 Same user cannot enter the same whiteboard room

In the project, I enforce unique usernames. When a user wants to join the whiteboard, I must check if his username has already existed in the whiteboard. If it has already existed, a pop-up window will remind the user that you need to change another username and try again.

2.4 Seek approval from manager

Once a user requests to enter the board, permission will be sent to the manager. The user only can enter the board when the manager approves. Figure 2 shows the pop-up window to ask for permission from manager.



Hi, manager. bob want to enter this board. Would you agree?

Figure 2: Seek Approval

2.5 Display online usernames

On the left side of GUI, a JList object displays all current alive users in the whiteboard. The list will change as users join and leave.

2.6 Synchronize the drawing board state

Users may connect and disconnect at any time. When a user joins the board, he will obtain the current state of the whiteboard. Thus, all active users can use drawing tools synchronously without delays.

3 Design

3.1 Java Remote Method Invocation

In this project, all clients will visit and edit a board at the same time. Whenever a client draws on the drawing board, the drawing board of other clients should be updated without delay. Considering that there is close communication between the server and the client, some latency is inevitable with the previous socket model. Thus, I use RMI architecture to implement communication between server and clients. I transfer array of byte among server and clients.

3.1.1 Remote Interface

Remote interface defines the contract between the client and the server. It also constitutes the base for both stub and skeleton. I define $IRemoteUser$ and $IRemoteBoard$ interfaces. One is used to manage clients in whiteboard project. Another is used to manage white board, including drawing shapes, monitoring state of the board, interacting with GUI.

$IRemoteBoard$ interface implements getter and setter functions for remote board, basic functions for drawing shapes used Java2D, check if username exit, remove a certain user from system, update active users' information, and manage information GUI and so on.

$IRemoteUser$ interface implements functions to manage users and synchronises the canvas state, such as adding users to the system. An error alert box pops up to interact with the user, show messages in the chat window, etc.

3.1.2 Servant Component

Servant components define specific functions to implement the remote interface. *RemoteDraw* class implements IRemoteBoard interface. *joinWhiteBoard* class and *CreateWhiteBoard* class implements IRemoteUser interface. Because these classes extend UnicastRemoteObject, it will export the remote board object automatically to the Java RMI after the object is initialized. So, it can receive incoming remote calls.

3.1.3 Server

In RMI, a server is the main driver to control connection with clients. *Server* class creates an instance of the RemoteDraw class and publishes its stub in the registry under the name " Whiteboard". The server will continue running as long as there are remote objects exported into the RMI.

3.1.4 Client

The client-side will look up RMI proxy depending on arguments, such as port number, IP address first. If it can find the "Whiteboard" proxy, all requests from GUI and functions to manage users and draw shapes will be transferred to RMI registry to perform.

3.2 Main Class

Figure 3 shows the UML class diagram of the project. Figure 4 shows an interaction diagram (sequence diagram).

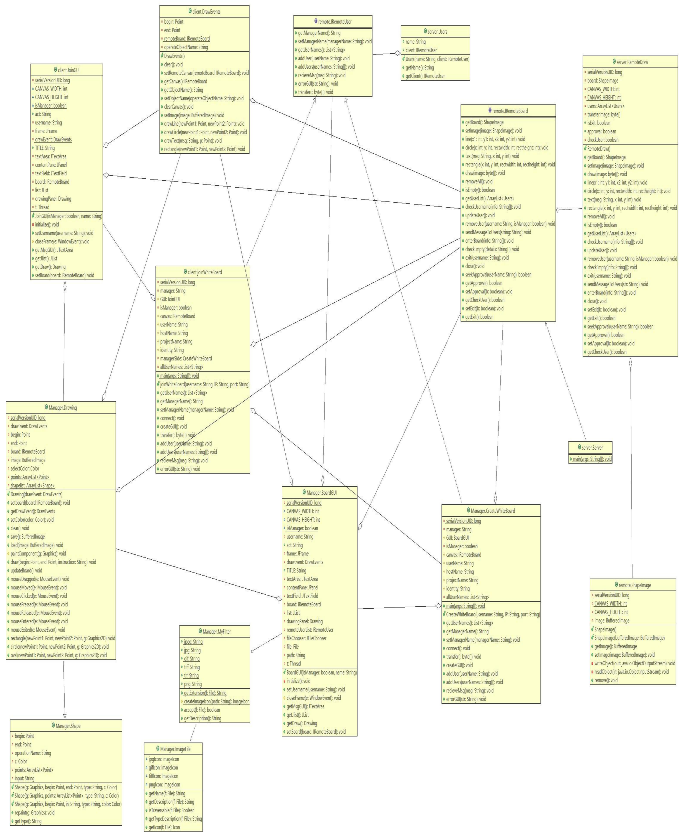


Figure 3: UML Class Diagram

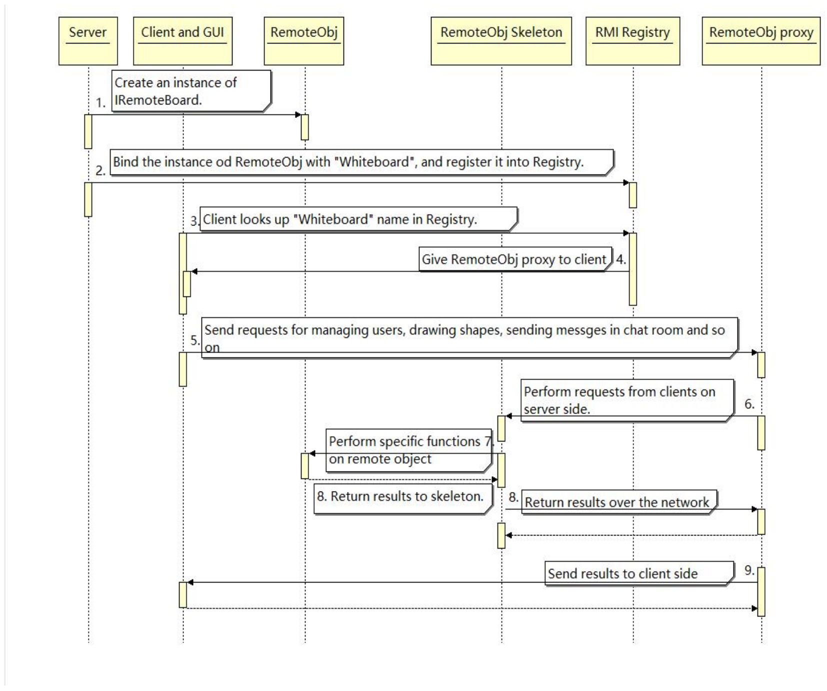

Server
Client and GUI
RemoteObj
RemoteObj Skeleton
RMI Registry
RemoteObj proxy
1. Create an instance of IRemoteBoard.
2. Bind the instance od RemoteObj with "Whiteboard", and register it into Registry.
3. Client looks up "Whiteboard" name in Registry.
Give RemoteObj proxy to client 4.
5. Send requests for managing users, drawing shapes, sending messges in chat room and so on
Perform requests from clients on server side. 6.
Perform specific functions on remote object 7.
8. Return results to skeleton.
8. Return results over the network
Send results to client side 9.

Figure 4: Sequence Diagram

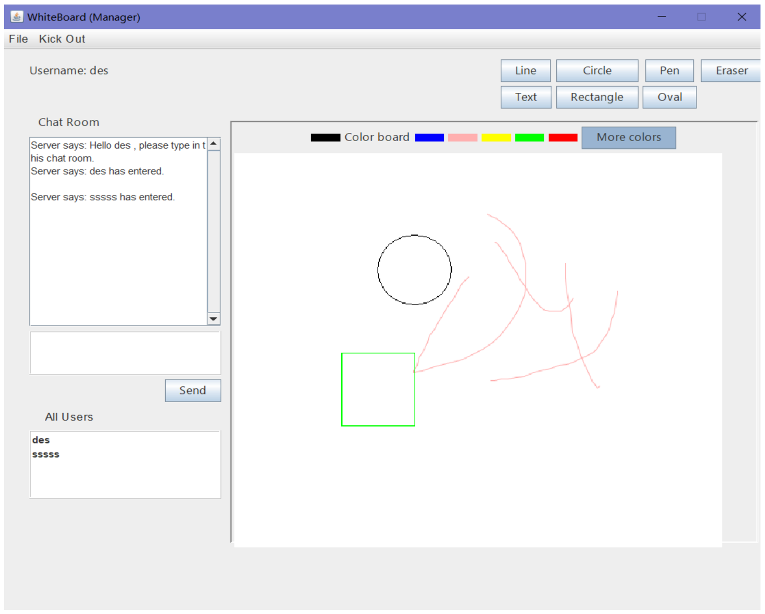
Server class creates an instance of the *RemoteDraw* class and publishes its stub in the registry under the name " Whiteboard". *Users* class defines a custom structure "Users". It contains username and remote client object.

IRemoteBoard and *IRemoteUser* interface are both extend Remote class. They define the contract and must be shared between client and server. *IRemoteBoard* interface defines functions about monitoring the state of the board, drawing basic shapes, interacting with GUI, seeking approval from manager when enter the board and so on. *IRemoteUser* interface defines functions to manage users list, sends a message to all other active users, and transfers data from client to server-side.

RemoteDraw class implements details of functions in the *IRemoteBoard* interface. It represents the remote object (skeleton). Client-side will call methods in it to perform requests. *joinWhiteBoard* class and *CreateWhiteBoard* class implements more specific details of functions in IRemoteUser interface. They also define the main function of the system, which parse arguments into IP address, port, and username.

ShapeImage class declares the default size of the canvas, obtain and remove canvas functions. It also defines functions to read or write the state of board into an object for sending to clients. *DrawEvents* class uses functions related to drawing in *RemoteDraw* class. And it also connects remote component to client-side. On top of *RemoteDraw* class, it modifies the function of drawing graphics to be more concise and easier to be called. *Drawing* class extends JPanel and implements *MouseListener*, *MouseMotionListener*. It implements the corresponding graphic drawing based on the currently used tool command obtained from the GUI. And add all shapes into a shape list for the next repaint. *Shape* class declares a list of shapes on the current board, which will be used to repaint the board among clients in a certain interval. It also stores essential variables for drawing shapes, such as begin point, endpoint, instruction name, colors.

MyFilter class extends *FileFilter* class, which implements common functions in *FileFilter* class. It only accepts some types of file, such as gif, jpg, or png files. *ImageFile* class extends FileView class. It defines how to obtain the current file's extension and image icons. *BoardGUI* class designs and implements the interface on manager side. It also detects button click events. For instance, store tool's name as instruction when user click on tool buttons; change current color during painting when user click another color on GUI. It also gives manager permission to manage files. Manager can save, save as, open, new and close the board, or kick out a user by typing his username. *JoinGUI* class implements the interface on client-side. It also detects button click events, such as changing colors, sending messages in chat window. Both *JoinGUI* and *BoardGUI* have a user list, chat window, a canvas for drawing, color chooser tool and basic buttons for painting. Figure 5, 6 shows interface on general users and manager side.



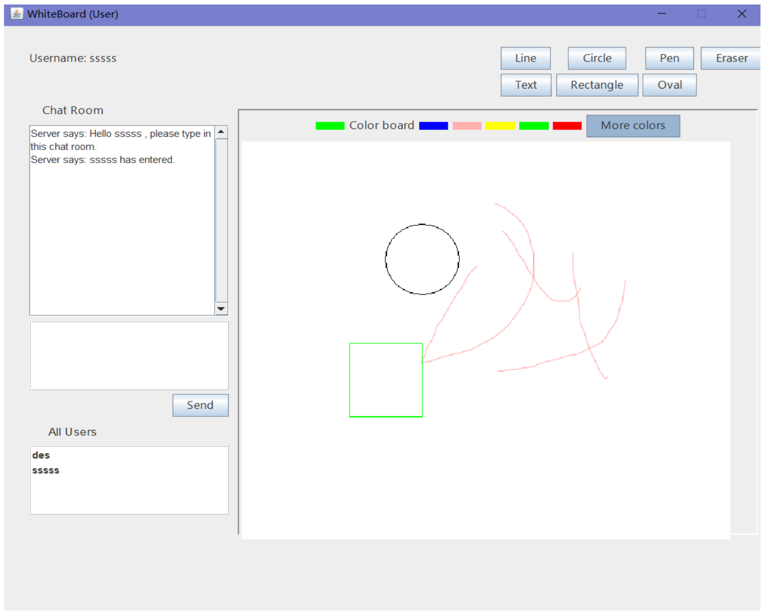
Username: des

Server says: Hello des , please type in t
his chat room.
Server says: des has entered.

Server says: sssss has entered.

des
sssss

Figure 5: Manager Interface



Username: sssss

Server says: Hello sssss , please type in
this chat room.
Server says: sssss has entered.

des
sssss

Figure 6: User Interface

3.3 Creativity Elements

3.3.1 Chat Window

On left side of GUI, there is a chat room to help all users communicate more conveniently. A notification will be displayed in the chat room when there is a change in the current online users, such as someone leaves the board, someone is kicked out by the manager, someone enter the board. Besides, all online users can type texts into the chat room in real-time.

3.3.2 A "File" Menu

There is a "File" button on the top of the GUI on the manager side(figure 7). It implements functions about managing the board information, such as open, save, save as, new and close the whiteboard. It is important to note that only the manager has the authority to perform these operations. I use a Boolean variable to identify whether the current client is the manager.

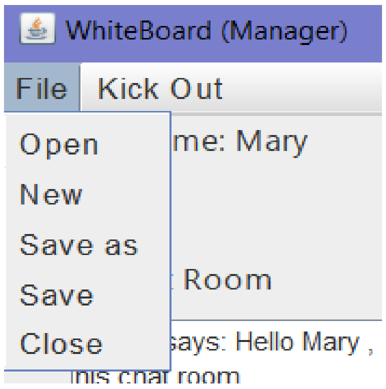


Figure 7: Menu

1) New

Once the manager clicks on "New" button in the file menu, a pop-up window will appear to ask if you need to save the current board first. If yes, you can choose a path to store the canvas. Otherwise, system will clear all traces from the drawing board, then create a new blank board.

2) Open

Once the manager clicks on "Open" button in the file menu, a pop-up window will appear to ask if you need to save the current board first. If yes, you can choose a path to store the canvas. Otherwise, a file chooser will appear for users to select an existed image to open in the board.

3) Save

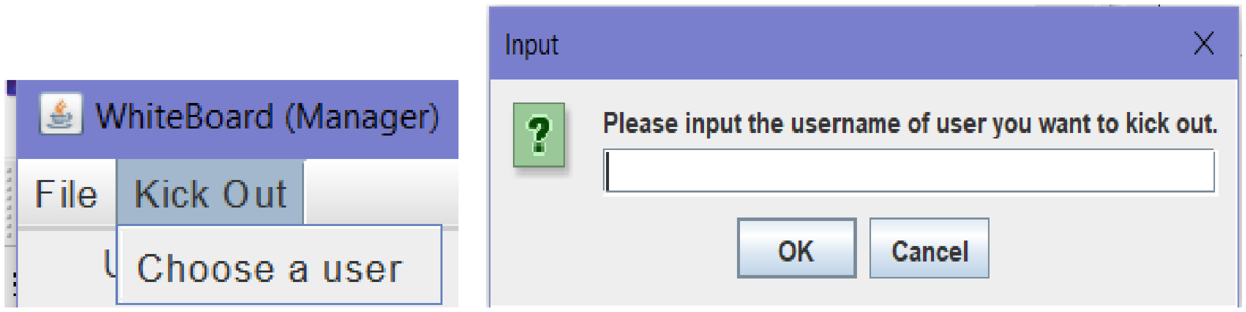
Once the manager clicks on "Save" button in the File menu, a file chooser window will appear. The user can select a path to store the current board as an image format, such as .jpg, .png. The default format to be saved is .jpg. After the user selects the storage path, each click on the "Save" button will store the current drawing board as a picture in that path. It will automatically overwrite the previously stored file object.

4) SaveAs

Once the manager clicks on "Save As" button in the File menu, a file chooser window will appear. The user can select a path to store the current board as an image format. The difference between "Save As" and "Save" is whether overwrite previous image. Each time user clicks on "Save As" button, a file window will appear to let user select the path. User can save board as an image in different path with different name.

3.3.3 Kick out

There is a "Kick out" button on the top of the GUI on the manager side (Figure 8). Only the manager has the right to kick out a user. Because each user has a unique username, the manager can type the user's username he wants to kick out to force the user to exit the board.



Please input the username of user you want to kick out.

Figure 8: Kick Out

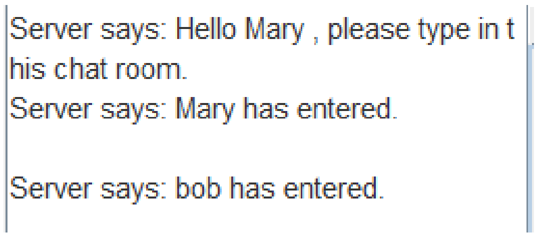
3.3.4 Pen and Eraser

During the drawing process, I store the coordinates of the points of the mouse's drawing track in an array. This enables continuous drawing. The eraser is based on the brush, the brush colour is changed to white and the width is set to 2.0f.

3.3.5 User interactions

A good system should interact well with the user, providing them with a readable explanation of the cause of the error and the next step to take during the interaction. In this system, I use some pop-up windows to inform users errors or instructions. A pop-up window will appear to inform user to change another name when his name exists in the board. There are also pop-up windows during system operation. For instance, the manager rejects the request to enter the board; a user is kicked out by the manager; the room is already created.

Besides, I use chat room to interact with all users. When there is a change in the current active users, a message from server will inform all users. Figure below shows the chat window when users enter.



```
Server says: Hello Mary , please type in t
his chat room.
Server says: Mary has entered.

Server says: bob has entered.
```

Figure 9: User Interface

4 Critical Analysis

4.1 RMI

Because RMI is part of Java's object oriented approach, it has some advantages over traditional methods.

- RMI is object oriented. It can pass whole objects as arguments and return values, not only predefined data types. RMI makes it simple to write remote Java servers and Java clients that access those servers.

- It stores the server objects in RMI Registry in advance, and each time a client needs to establish a connection, they only need to look up the server they want to connect to in RMI Registry. Only one registration to the server is required. So users can establish connections themselves.
- All interactions between the server and the clients are done by RMI Registry. Latency problems of sockets connections are avoided.
- The core method of the main operation can be passed through RemoteObj proxy and RemoteObj Skeleton. It can reduce operational burden on the client and server side. The code of RMI is highly extensible and the structure of the code is very concise.

Followings are disadvantages over sockets.

- Remote method calls are made in RMI by using a Stub object on the client side as the remote interface. Each remote method has a method signature. If a method is executed on the server, but no matching signature is added to the remote interface (stub), then the new method cannot be called by the RMI client.
- Because RMI provides less control on the data being transmitted across the network, it may appear computation and communication overheads.
- It is less efficient than Socket objects. If you plan to visit data on server, you need to define a specialized method in remote interface. Besides, RMI is hard to keep security during connections.

4.2 Kick Out

In this system, the manager can type username to kick out a particular user. However, I don't think this is enough in terms of interactivity with the users. Thus, I came up with an improvement. When the manager clicks on "Kick out: button on menu, it will show a list of current alive users. The manager just need to click one of them to kick out users. It may be more convenient for manager to manage users.

5 Conclusion

This project uses RMI mode to implement shared whiteboard and support for concurrent clients operations. In this project, I learn the core idea of RMI and implement it on my own. I also learn how to implement color chooser, file chooser and many components in Java Swing. And I use many pop-up windows to enhance the user interaction experience. Besides, this project allows me to consider all exception cases and how to handle them. I also learn how to save images and transfer image information and so on.