Eric Roberts
CS 106A

Handout #50
February 19, 2010

Assignment #5—FacePamphlet 2.0

The original version of FacePamphlet was designed by Mehran Sahami.

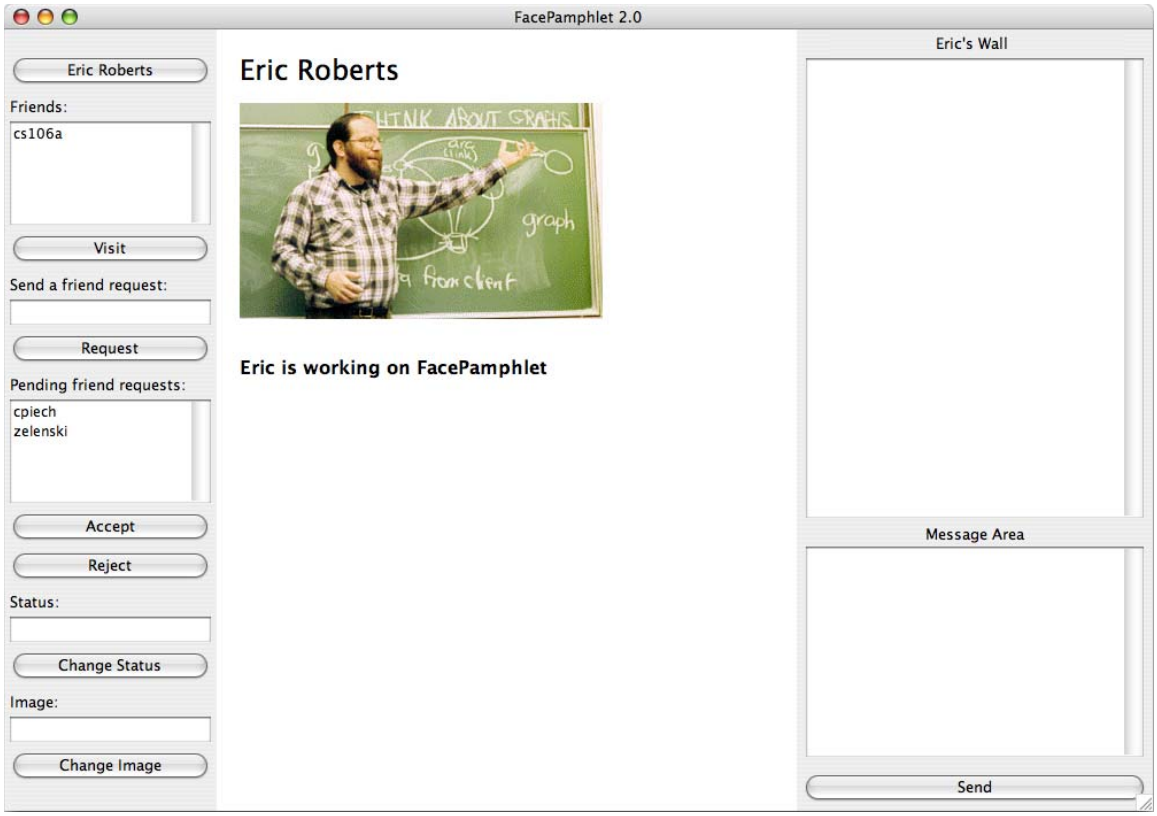
Due: Monday, March 1

This assignment has two goals. The first is to give you an opportunity to use Java interactors to create an interactive application complete with buttons and various kinds of text fields, along with a graphical display. The second goal is to give you the chance to create a CS 106A social network—much like Facebook—designed using the **client-server model** in which a client application on one machine talks to a server running on another. Your task here is to write the client side of an application that does at least a few of the things that Facebook does. Your application will, of course, be a somewhat simpler version—a pamphlet-sized application instead of a book.

Overview of the FacePamphlet application

When you get the full application working, the FacePamphlet application will present the user with a graphical user interface that looks like the one shown in Figure 1. In this example, I have already logged in as **eroberts**, and set my image and my status. All of you, including your section leaders, have an account specified by your SUID and a new password just for FacePamphlet. The display also tells me that I have one friend—the virtual **cs106a** user who is automatically a friend to everyone in the class. I also have pending friend requests from **cpiech** (Chris Piech) and **zelenski** (Julie Zelenski). As of now, my "wall" at the right is empty, because no one has yet written anything on it.

Figure 1. User interface for the FacePamphlet application



The interactors along the left side of the window control most of the operation of my FacePamphlet application. I could, for example, change my status by entering new data into the status field and clicking the **Change Status** button. If I want to report that I am now preparing for lecture, I can type

preparing for lecture

into the text field and then click the **Change Status** button to bring me to Figure 2.

As another example, I can accept Chris Piech's friend request by selecting **cpiech** in the list and then clicking on the **Accept** button. When I do so, **cpiech** disappears from my list of pending friend requests and shows up in my list of friends, as shown in Figure 3.

More interestingly, once **cpiech** is in my list of friends, I can visit Chris's page by clicking on his identifier in the friend's list and then clicking the **Visit** button. This brings me to the state shown in Figure 4 at the bottom of the next page. I can now see Chris's picture and status; I also see Chris's wall, where some student has posted a message. The controls along the left side of the window, however, are still mine, as the name of the button at the top makes clear. I could visit someone else's page, accept or reject new friend requests, or update my status or image.

Now that I'm visiting Chris's page, I can post a message on his wall. To do so, I simply type a message into the message area at the bottom of the window, and click the **Send** button. When I do, the message gets appended to Chris's wall, and the message area itself is cleared. The situation before and after clicking **Send** is illustrated on page 4 in Figures 5 and 6. The signature line showing that the message came from me is added automatically by the repository code.

Figure 2. Changing status

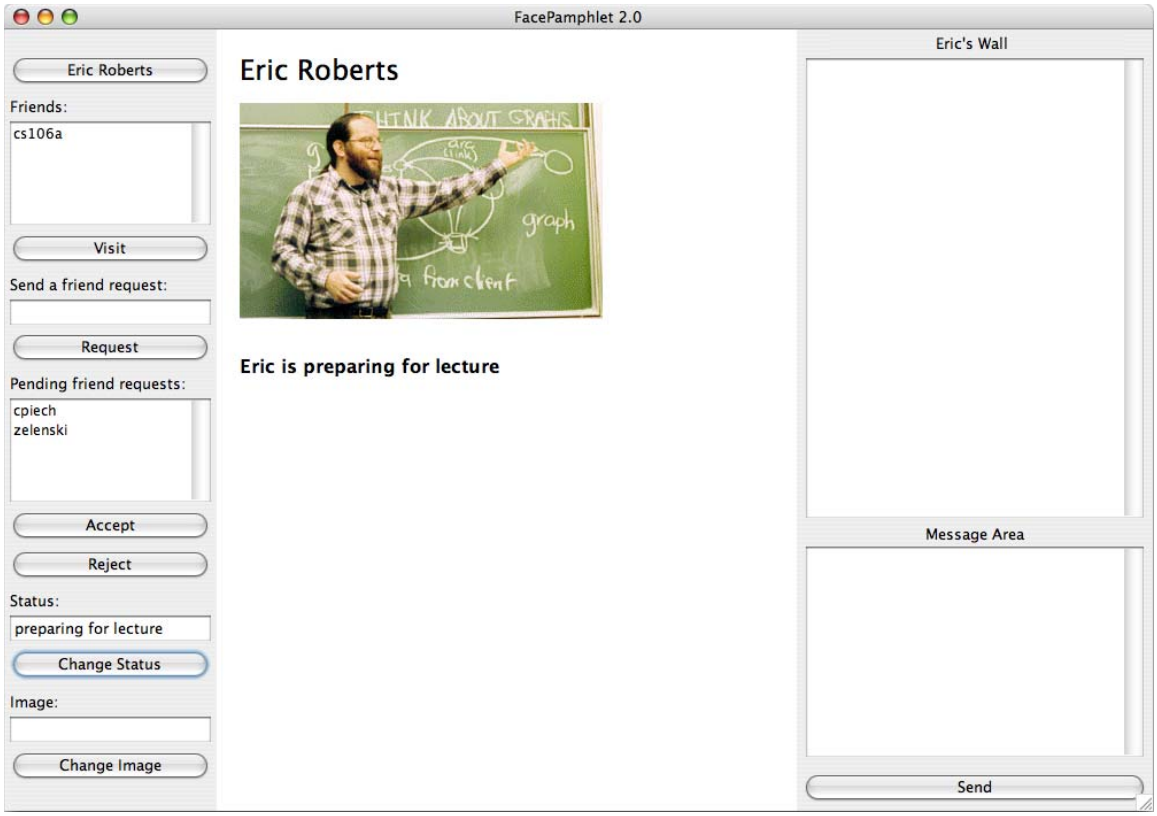


Figure 3. Accepting a friend request

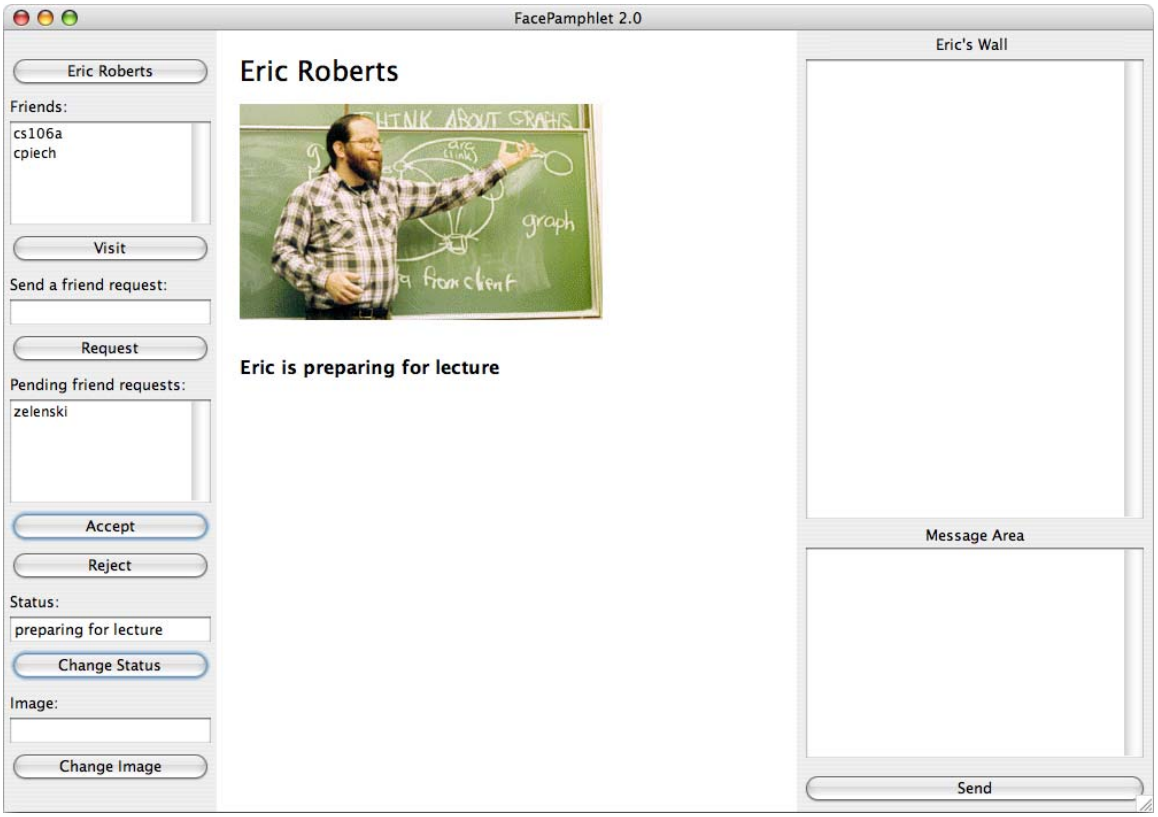


Figure 4. Visiting pages of a friend

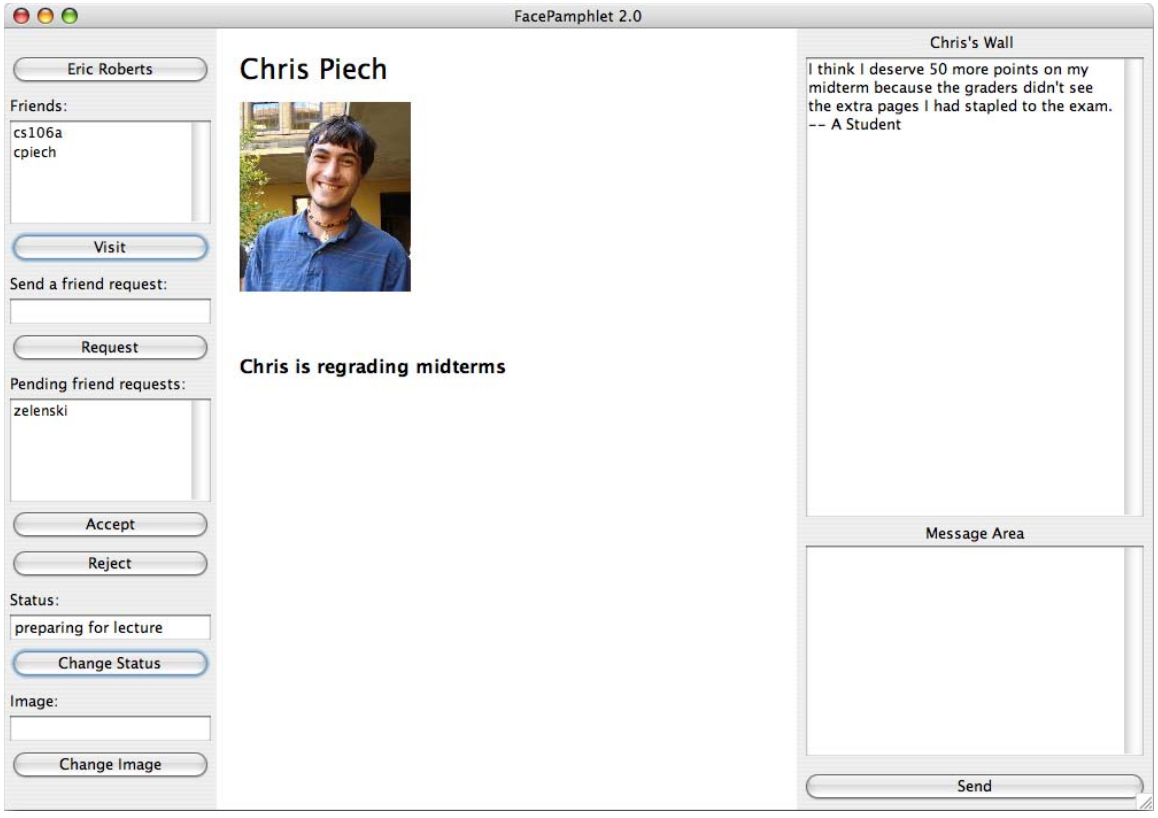


Figure 5. Writing on someone's wall (before clicking Send)

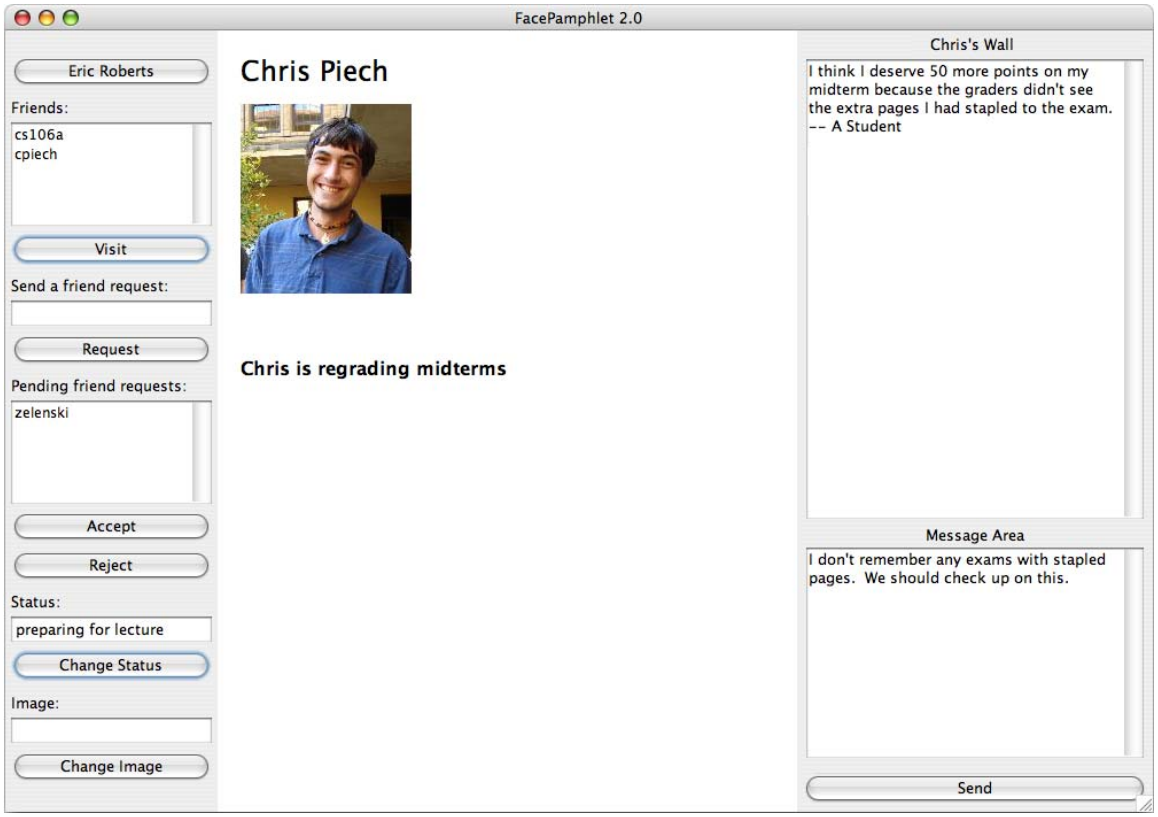
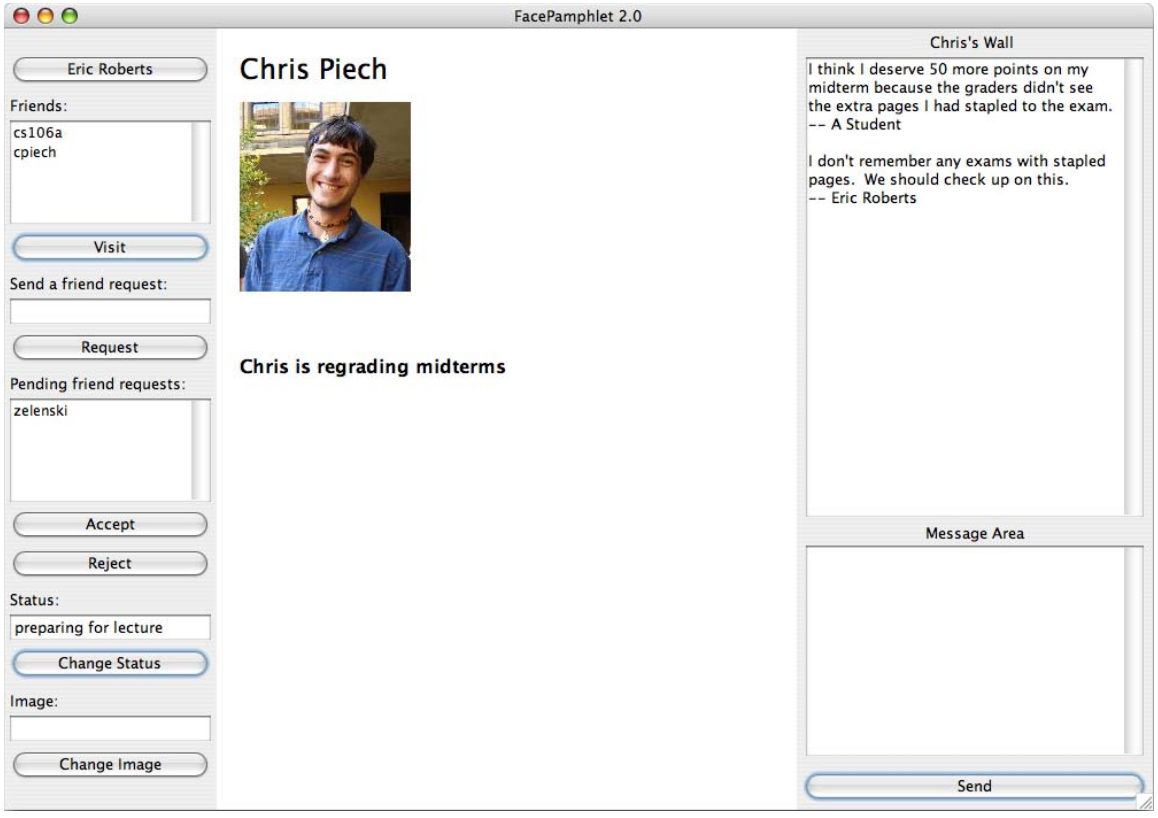


Figure 6. Writing on someone's wall (after clicking Send)



Client-server architectures

The most exciting thing about the FacePamphlet 2.0 application is that it actually does its work over the network, making it possible for all of you to interact, just as you do on social network sites like Facebook. The image and status on Chris's page are the ones that he put there. When I visit Chris's page, my copy of the FacePamphlet program loads that data from the network. Similarly, when I update my status or write a message on someone's wall, that information is stored on the network so that other users can see it.

What makes all of this possible is an implementation strategy called a **client-server architecture,** in which a single machine—the **server**—is responsible for storing all of the data in its file system, so that it persists even when no applications are running. Each user runs a FacePamphlet **client**, which puts up the user interface shown in Figures 1 through 6 and then communicates with the server to store and retrieve information. The basic structure of the client-server architecture is illustrated in Figure 7. Note that there can be many different clients, each of which manages one user interface window, but only one server, which handles all the updates to the hard disk on the server machine.

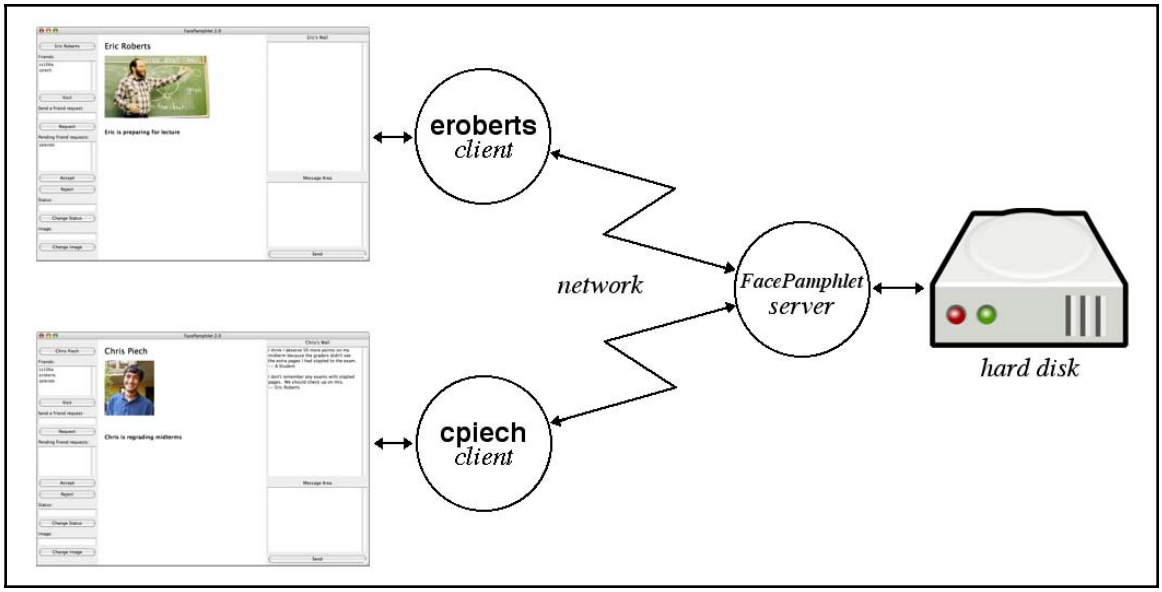
The files in the starter project

Given all the things it does, this assignment sounds huge, but the amount of code you have to write is actually quite small. We've written the code for the server and for the network-communication side of the client. To make things even simpler, we've given you the code for a few of the classes you need for the user interface as well. The only code you have to write is in the **`FacePamphlet`** class.

The starter project contains **`.java`** files for the following classes:

- **`FacePamphlet`**—This is the program class that runs the FacePamphlet application. In the form that we give it to you, it already contains a few bits and pieces of code, but most of it is yours to write.
- **`FPConstants`**—This interface defines a set of constants that you can use in the rest of the program, just as the **`YahtzeeConstants`** interface did in Assignment #4. You need to be familiar with the constants this interface exports and use those names instead of their values when you write your code.

Figure 7. The client-server architecture of FacePamphlet



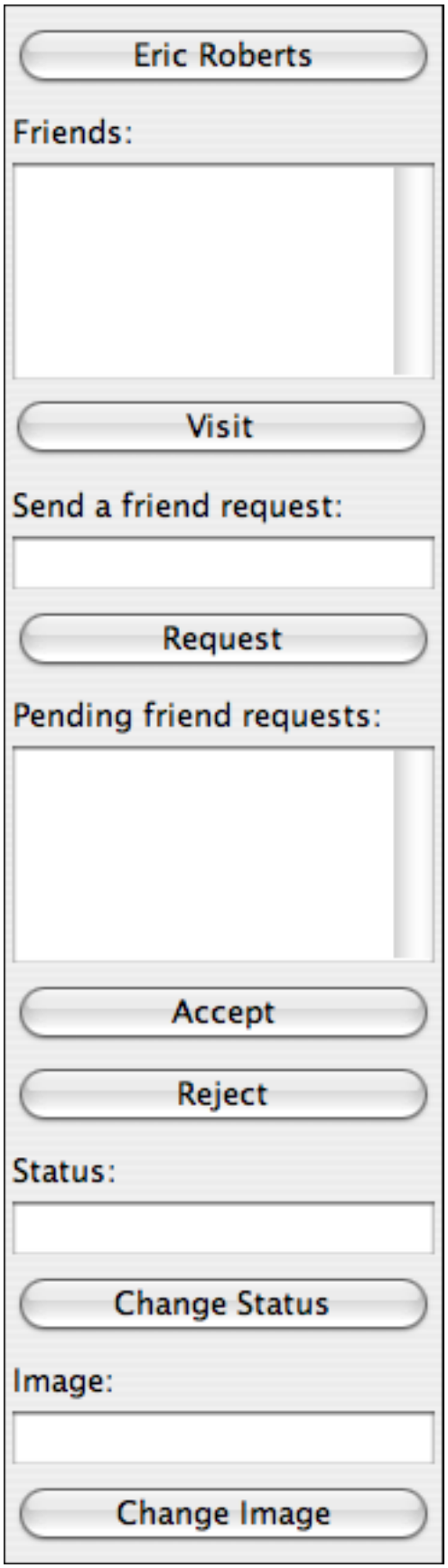
- **`FPRepository`**—This class implements the client side of the network connection to the FacePamphlet server. Its methods (described in a new handout on Monday) allow you to look up information about yourself and your friends and to modify your own data.
- **`FPRepositoryStub`**—This class is a subclass of **`FPRepository`** that simulates the operation of the repository without actually going to the network. As described in the milestone list later in this handout, the general strategy is to debug your program using the stub repository and to move to the real one when you get everything working.
- **`FPScrollableList`**—This class implements a scrollable list. Two instances of this class appear in the user interface, both on the left side of the window. The first shows the list of friends, and the second shows the pending friend requests. The methods exported by this class are described briefly in Figure 8.
- **`FPScrollableTextArea`**—This class implements a scrollable text area. Two instances of this class appear on the right side of the window. The first is the "wall" for the user you are currently visiting, and the second is the message area in which you enter new messages to send. The methods exported by this class are outlined in Figure 9.

Figure 8. Public entries in the `FPScrollableList` class

`public FPScrollableList()` This constructor creates an empty list with a vertical scrollbar.
`public void clear()` Removes all the entries in the list to make it empty.
`public void add(String name)` Adds a new name to the bottom of the list.
`public String getSelectedName()` Returns the selected name in the list, or **`null`** if no name is selected.
`public void setActionCommand(String command)` Sets the action command triggered when the user double clicks on a name.
`public String getActionCommand()` Returns the action command triggered when the user double clicks on a name.
`public void addActionListener(ActionListener listener)` Adds an action listener to the list, which can then respond to double clicks.
`public void removeActionListener(ActionListener listener)` Removes the specified action listener.

Figure 9. Public entries in the `FPScrollableTextArea` class

`public FPScrollableTextArea()` This constructor creates an empty text area with a vertical scrollbar.
`public void setText(String text)` Sets the text to the specified string, which may contain end-of-line characters.
`public String getText()` Returns the text in the text area.
`public void setEditable(boolean flag)` Indicates whether the user can change the text (**`true`** means yes; **`false`** means no).
`public boolean isEditable()` Returns **`true`** if this text area is editable, and **`false`** otherwise.



Milestone 1: Creating the west control panel

Your first task in this assignment is to add interactors to the control panel that appears at the left of the window, which is the **`WEST`** region of the program layout. The contents of this panel appear at the left of this page. The two areas containing scrollbars—one for the list of friends and one for the pending friend requests—are **`FPScrollableList`** objects as described on the previous page. The others are either **`JButton`**s, **`JLabel`**s, or **`JTextField`**s, as described in Chapter 10 of the text. Your task here is simply to create each of the interactors and add them to the **`WEST`** region.

The only bit of subtlety here is deciding whether you need to store the interactor in an instance variable so that you can make use of it later on. The **`JLabel`**s on this panel, for example, are simply informational text; once you create them, you never need to refer to them again. The two **`FPScrollableList`** objects and the **`JTextField`**s, by contrast, need to be stored in instance variables because you will need to call their methods to retrieve or change their contents. The **`JButton`**s can be managed either way. If all you need to know is the name of the button when an action event occurs, you can get away with creating the buttons as you go, relying on the fact that the action events they generate include the button label as their action command.

The one button that needs to be an instance variable is the one at the top of the window. The label on this button is the full name of the current user, which you will eventually get from the repository. At that point, you will need to call **`setText`** on that **`JButton`** object, which means that you will have to have it around.

The first milestone, therefore, is simply to add all of the interactors to the west side panel, even though they don't do anything as yet. This phase involves adding any new instance variables you need, initializing those instance variables to contain the interactors, and then adding them to the **`WEST`** panel using calls like

```
add(new JLabel("Friends:"), WEST);
```

These calls must all happen during the execution of the **`init`** method, but should probably not be directly within it. Remember that decomposition is your friend. At the very least, you should define a method called **`initWestPanel`** (or something like that) that sets up this side panel. You could, however, decompose this further to create the logically connected parts of this side panel.

Milestone 2: The east control panel

The panel on the east side is in most respects simpler than the one on the west side, but there are a couple of wrinkles. The most obvious difference is that the two large interactors on this side are **`FBScrollableTextArea`**s. The top one is used to display the contents of someone's wall files and is therefore not directly editable by the current user. The bottom one, however, is used to create messages, which means that the user must be able to type into it. When you create the bottom one, you must therefore call

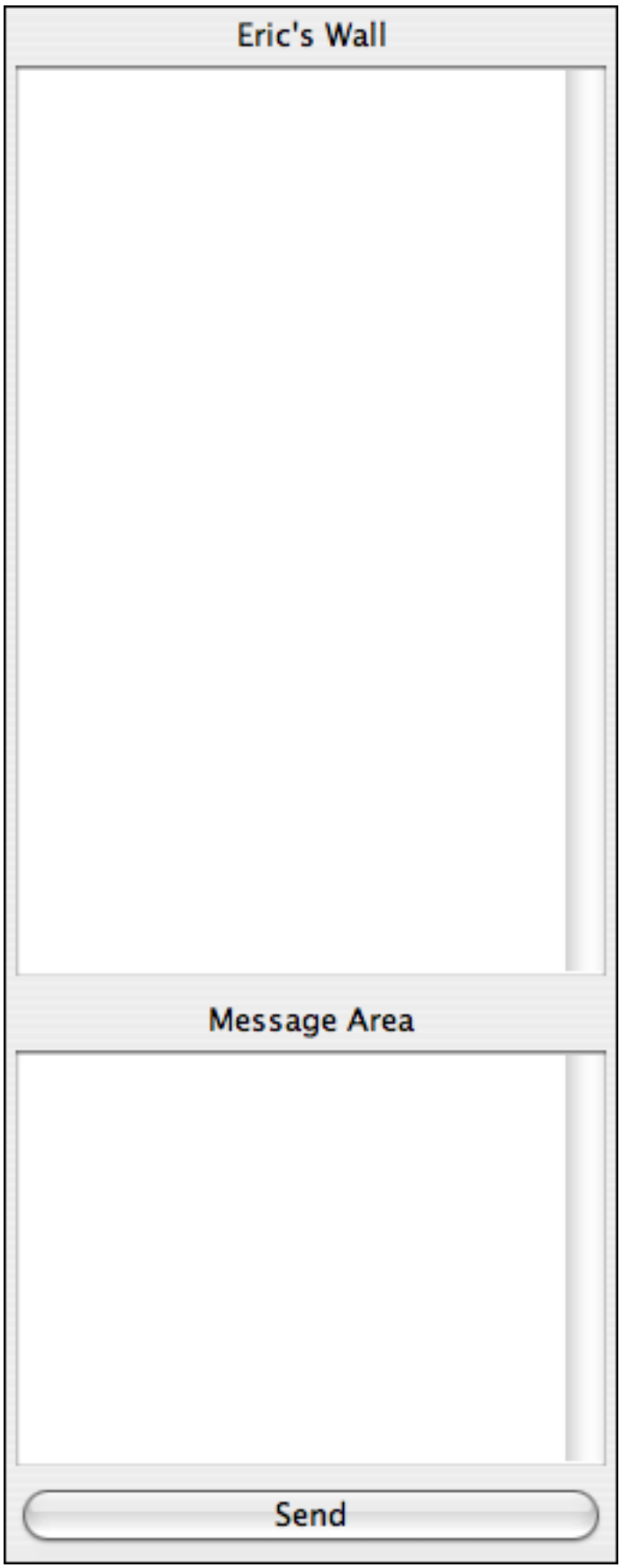
```
setEditable(true);
```

to ensure that the area functions correctly.

Another small wrinkle is that the **`JLabel`**s on this side are centered rather than aligned on the left, which is the default behavior for the **`JLabel`** class. To center the label, you need to set its horizontal alignment property to the constant **`JLabel.CENTER`**. You can set that property either by calling the method **`setHorizontalAlignment`** on the label or by passing **`JLabel.CENTER`** as a second argument to the **`JLabel`** constructor.

The one other thing you need to think about is the fact that the text of the **`JLabel`** at the top of the page changes depending on whose wall you're seeing. When I started up my FacePamphlet application, the text on that label was **`Eric's Wall`**. When I visited Chris's profile later, it read **`Chris's Wall`**. The fact that you need to change the text of the **`JLabel`** means that you had better keep it around in an instance variable.

An additional requirement is that the title of the wall should include the user's first name. As you will see later, you can get the full name of any user from the repository, which will give you back a string like **`"Eric Roberts"`** but not the first name by itself. While you're thinking about the problem is a good time to write yourself a method that extracts the first name from a full one. You'll need it again for the graphical display.



Milestone 3: Adding action listeners

At this point in the program, you have very elaborate side panels on both the west and east sides of the window, but none of the interactors actually do anything. Before you can make your application work, you need to make sure that the buttons (and the other interactors as well) generate action events when they are activated. Moreover, you have to define an **actionPerformed** method in your program so that it can respond correctly when those events occur. You can add an action listener to all the buttons in the window just by calling

```
addActionListeners();
```

in the **init** method. For the **JTextField** and **FPScrollableList** interactors, however, you need to add action listeners explicitly. For example, if the interactor for your list of friends is called **friendList**, you can set things up so that double-clicking on a name in the list does the same thing as selecting a name and then clicking on the **Visit** button by adding the following statements to your initialization code:

```
friendList.addActionListener(this);
friendList.setActionCommand("Visit");
```

The more interesting part of this milestone is writing the code for **actionPerformed**. This method needs to look at the action command in the event and determine what button was activated. Your implementation then needs to take some response to that action.

Before you actually go to all the trouble to figure out what repository commands you need to call in order to set everything up, it's useful to make sure that your action-handling code works at all. One way you can accomplish this goal is to print out a message in response to every button that the user presses. That way, you know that at least that part of the your application is working.

One of the features of the assignment that makes this phase easy to do is that the starter file for the **FacePamphlet** class extends **ConsoleProgram** instead of **GraphicsProgram**, which it will have to do in the fullness of time. That means that you have a console available to you on which to trace the operation of your code. Suppose, for example, that you implement the **actionPerformed** method like this:

```
public void actionPerformed(ActionEvent e) {
   println(e.getActionCommand());
}
```

That implementation will write out every action command that gets generated as you click on buttons (or the other interactors that simulate button presses). If nothing appears on the console when you click those buttons, you've got a problem that you need to solve before you continue.

Printing out the action command string, however, does not really help you advance toward the final implementation of your program. When you click on certain buttons, you want to perform an action that often involves data that may reside in other interactors in the control panels. If you click on the **Change Status** button, for example, you will want to read the contents of the status text field. It would be great if you could have your implementation of **actionPerformed**, on detecting an action event generated by the **Change Status** button, call a method named **changeMyStatus** (or something similar), passing in the contents of the text field. For the purposes of Milestone 3, that method might be something as simple as

```
public void changeMyStatus(String newStatus) {
   println("Change my status to " + newStatus);
}
```

For the next milestone, you can change the implementation of this method—and the similar ones for all the other buttons—to make the correct calls on the repository.

Milestone 4: Add the repository calls

Most of the interactors on the window have the effect of sending a message from the client across the network to the server, possibly waiting for some kind of response in return. The **FPRepository** class exports a variety of methods for communicating with the server. These methods will be described in more detail in an additional handout next Monday, but the general idea is easy to illustrate by example.

Continuing on from the example in the preceding section, suppose that you wanted to update the implementation of **changeMyStatus** so that it performs the calls necessary to make the changes on the server. To do so, the code must call the **setMyProperty** method in the repository class, passing in a key for the property you want to change along with the new value of that property. The key for the status property is stored in the constant **STATUS_KEY** in the **FPConstants** interface, so the necessary code is

```
public void changeMyStatus(String newStatus) {
   repository.setMyProperty(STATUS_KEY, newStatus);
}
```

The variable **repository** is an instance variable that is already defined as part of the starter code. In the initial version, it is of type **FPRepositoryStub**, which implements a simplified version of the class that performs these operations locally rather than going out to the network. That's not as exciting as the real version, but should be enough to get things tested.

Unfortunately, the code for **changeMyStatus** as it currently stands doesn't really test whether the operation succeeded. For the purposes of Milestone 4, it would be better if you could also make a call to **getMyProperty** to see whether the change was recorded correctly. A more effective test implementation therefore looks like this:

```
public void changeMyStatus(String newStatus) {
   repository.setMyProperty(STATUS_KEY, newStatus);
   println("status = " + repository.getMyProperty(STATUS_KEY));
}
```

That second line will have to go away in the final implementation, but you need it in place for testing.

Milestone 5: Building the graphics display

The next step in moving toward the final implementation of the FacePamphlet application is to implement the graphical display that occupies the center of the window. This display consists of three graphical objects that you can see in the examples shown in Figures 1 through 6:

- A **GLabel** that displays the name of the person that you are viewing
- A **GImage** that shows their picture
- A **GLabel** that indicates that person's status

In addition to these three objects you can see in the examples, there is a fourth that appears only when the application wants to report an error or some similar message to the user:

- A **`GLabel`** at the bottom of the window that displays a user message

The locations, sizes, and fonts for these four objects are defined as constants in the **`FPConstants`** interface. The comments associated with those constants—particularly the comments associated with **`IMAGE_HEIGHT`**—define in detail how you should display each of the objects.

The first change you need to make in the program is to have the **`FacePamphlet`** class extend **`GraphicsProgram`** instead of **`ConsoleProgram`**. Once you've done that, you can put the various graphical objects in the right places on the window. Given that there are only four of them, this problem is much easier than the Breakout assignment, where you needed to create a whole bunch of bricks, a paddle, and a ball. Moreover, given that these objects don't move, there isn't really a need for an animation loop. Everything happens in response to events in the graphical user interface.

The values stored in these graphical objects, however, will change. If you reset your status, the contents of the **`GLabel`** have to change as well. If you change your image, you will have to update the contents of the **`GImage`**, but may also have to adjust its scaling so that it fits in the available space. And, once you move to the networked version of the program, it's possible that values change even when you yourself didn't take any action, since someone else might have written on your wall or updated the contents of their own profile while you happened to be visiting it.

The simplest strategy for updating the contents of the graphics window is to regenerate everything from scratch whenever you perform any user action. When you click on any button, for example, your code can throw away everything on the graphics window and then build it up again from scratch, going to the repository to make sure that it has the most up-to-date value of every field. This update operation extends beyond the graphical objects in the center window, since you also need to update the contents of the friends list, the pending request list, and the wall of the person that you are visiting. The repository has methods for getting the necessary value for each of these interactors; when an update occurs, you have to go through and call these methods to update the contents of the interactors that appear on the screen.

Milestone 6: Shift over to the networked repository

If you get everything working with the stub version of the repository, you're ready to move up to the networked version. If you're lucky, the only thing you need to do is change the line at the beginning of the **`init`** method—the one line of actual code we gave you to get started—so that it creates a real **`FPRepository`** object rather than a stub version. Calling the **`FPRepository`** constructor prompts you for a name and password (not your SUNET password, but one that we will mail to you over the weekend) and uses that information to make a connection to the server. Assuming that works, you can put up a picture of yourself, keep your status current, invite and accept (or reject) invitations from friends, and start interacting in our local version of the Facebook world!