

Assignment 5: FacePamphlet

The original assignment was written by Mehran Sahami and then revised by Eric Roberts (who also wrote the majority of this handout, save for some minor edits by Jerry.)

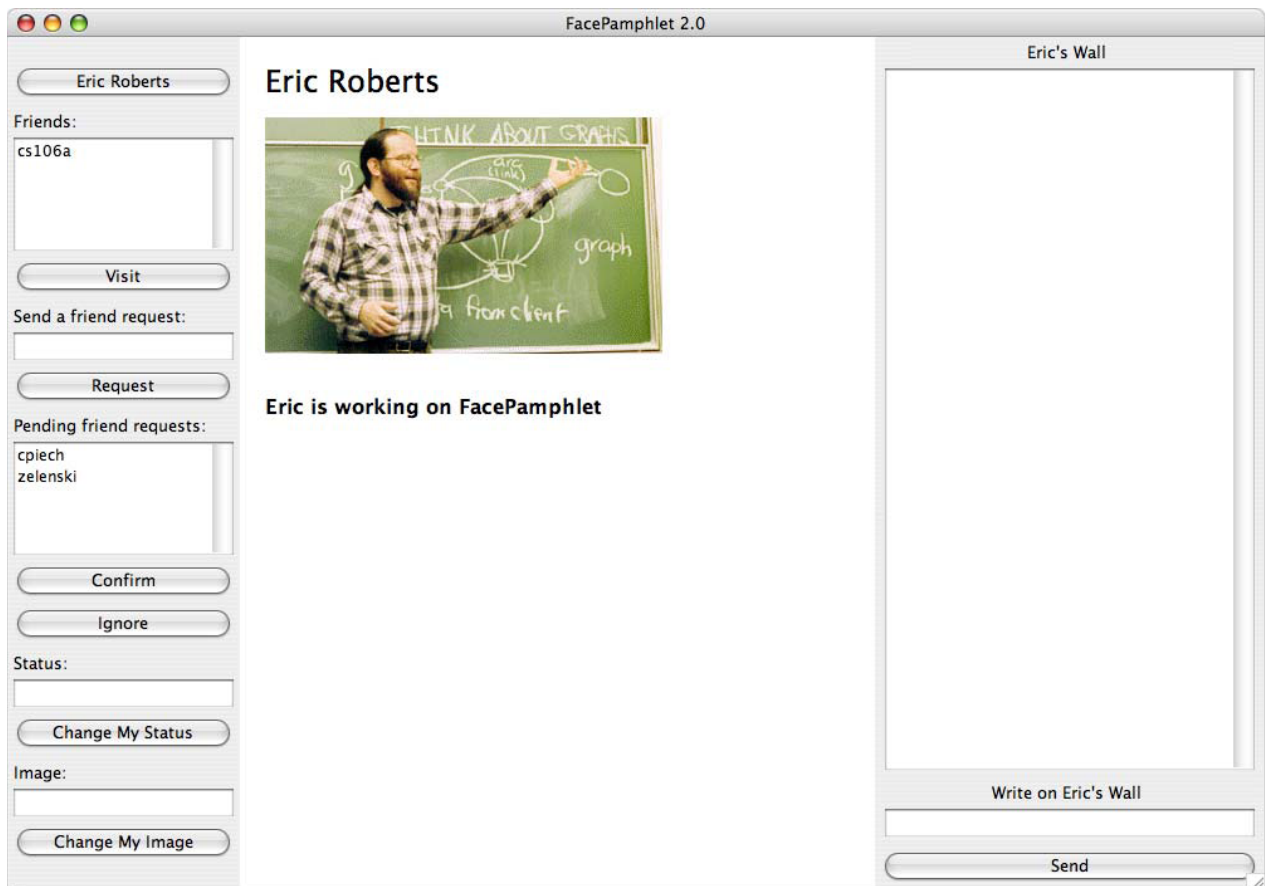
This assignment has two goals. The first is to give you an opportunity to implement an interactive application that uses a graphical user interface, complete with buttons and various kinds of text fields. The second is to give you the chance to write a social networking application, similar to Facebook, that lets users from the CS106A community to communicate with one another over the network. Your application will, of course, be a simpler version—a pamphlet-sized application instead of a book.

Due Monday, May 24th at 5:00 p.m.

Overview of the FacePamphlet application

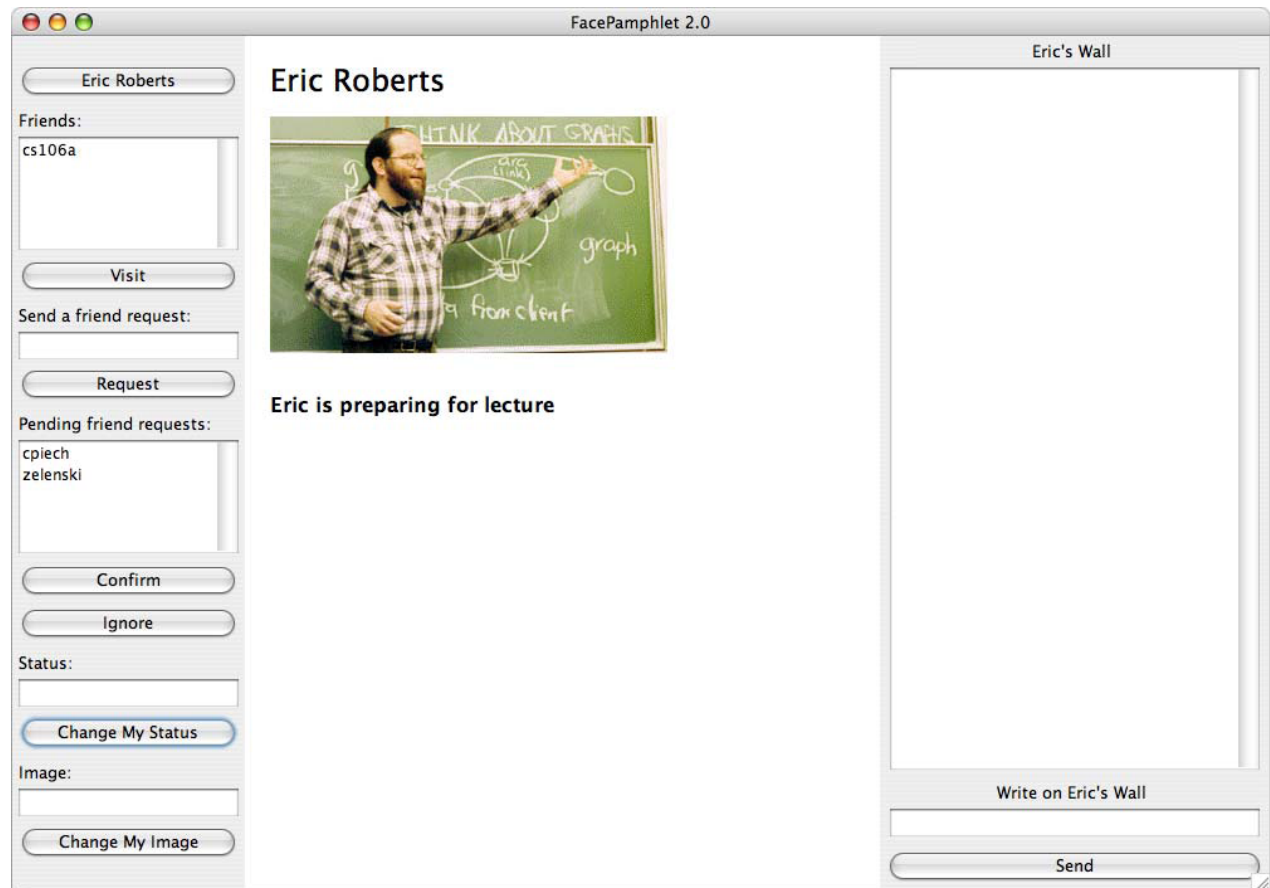
When you get the entire program working, the FacePamphlet application presents a graphical user interface that looks like the one shown in Figure 1. In this example, I (this is Eric Roberts speaking, not Jerry) have already logged in as **eroberts**, and have set my image and status. The display tells 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: Fully Operational FacePamphlet



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 My 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 My Status** button to bring me to the state shown in Figure 2.

Figure 2: Changing Status



As another example, I can accept Chris Piech's friend request by selecting **cpiech** in the list and then clicking on the **Confirm** button. When I do so, two things happen. First, **cpiech** moves from the list of pending friend requests to the friend list, where it appears before **cs106a** in alphabetical order. Second, a note confirming that **cpiech** is now a friend appears in the **message area** at the bottom of the window, as shown in Figure 3. The message area tells the user about changes in status and reports any errors that occur.

Once Chris is in my list of friends, I can visit his page by selecting **cpiech** in the friend's list and clicking the **Visit** button. This brings me to the state shown in Figure 4. I can now see Chris's picture and status. I also see Chris's wall, where some student has already posted a message. The controls along the left side, however, are still mine, as the name of the button at the top indicates. In the rest of this handout, the term **home user** refers to the user who is logged into the repository; **visit user**, by contrast, refers to the user whose information happens to appear in the center and east panels of the display. Thus, in Figure 4, the home user is **eroberts** and the visit user is **cpiech**.

Figure 3: Confirming a Friend Request

3

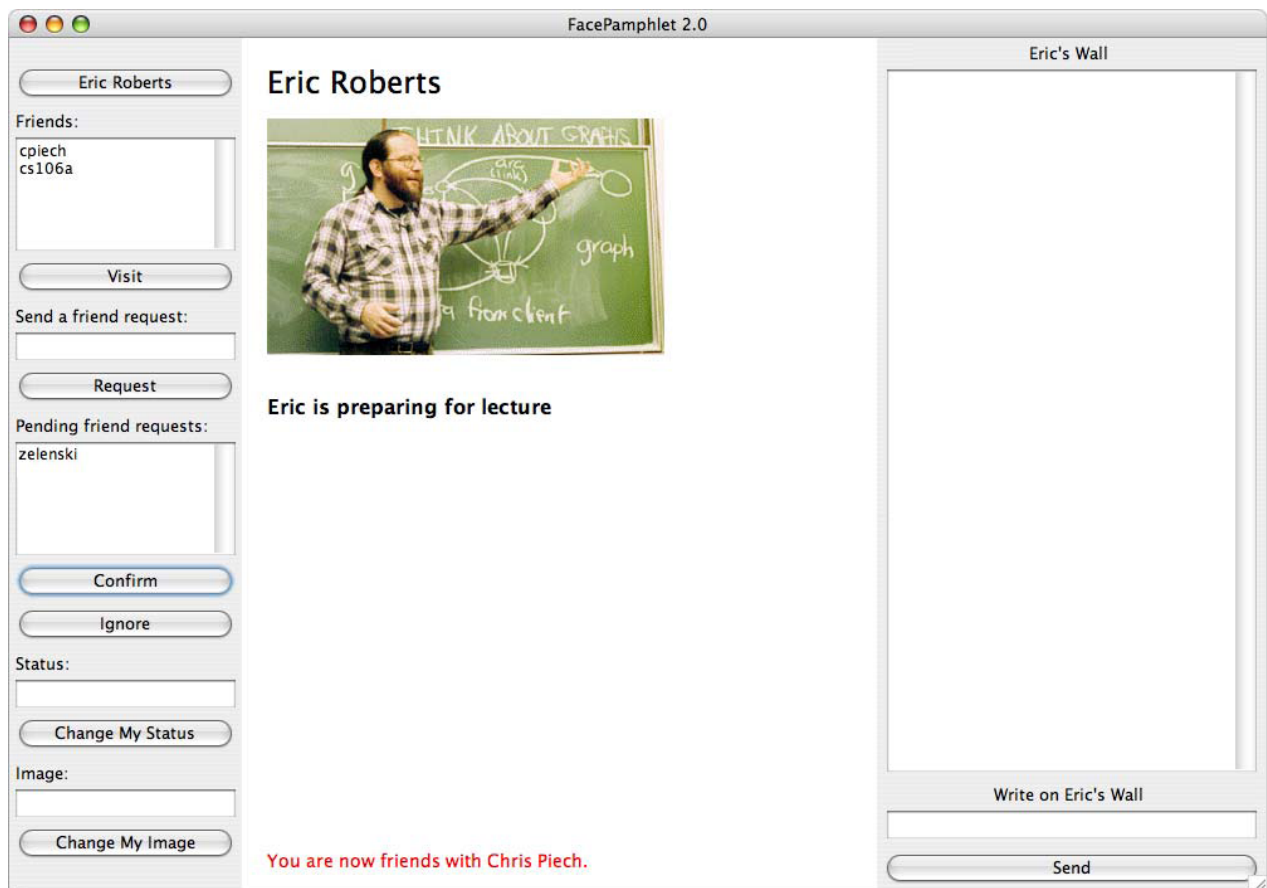


Figure 4: Visiting a Friend's Page

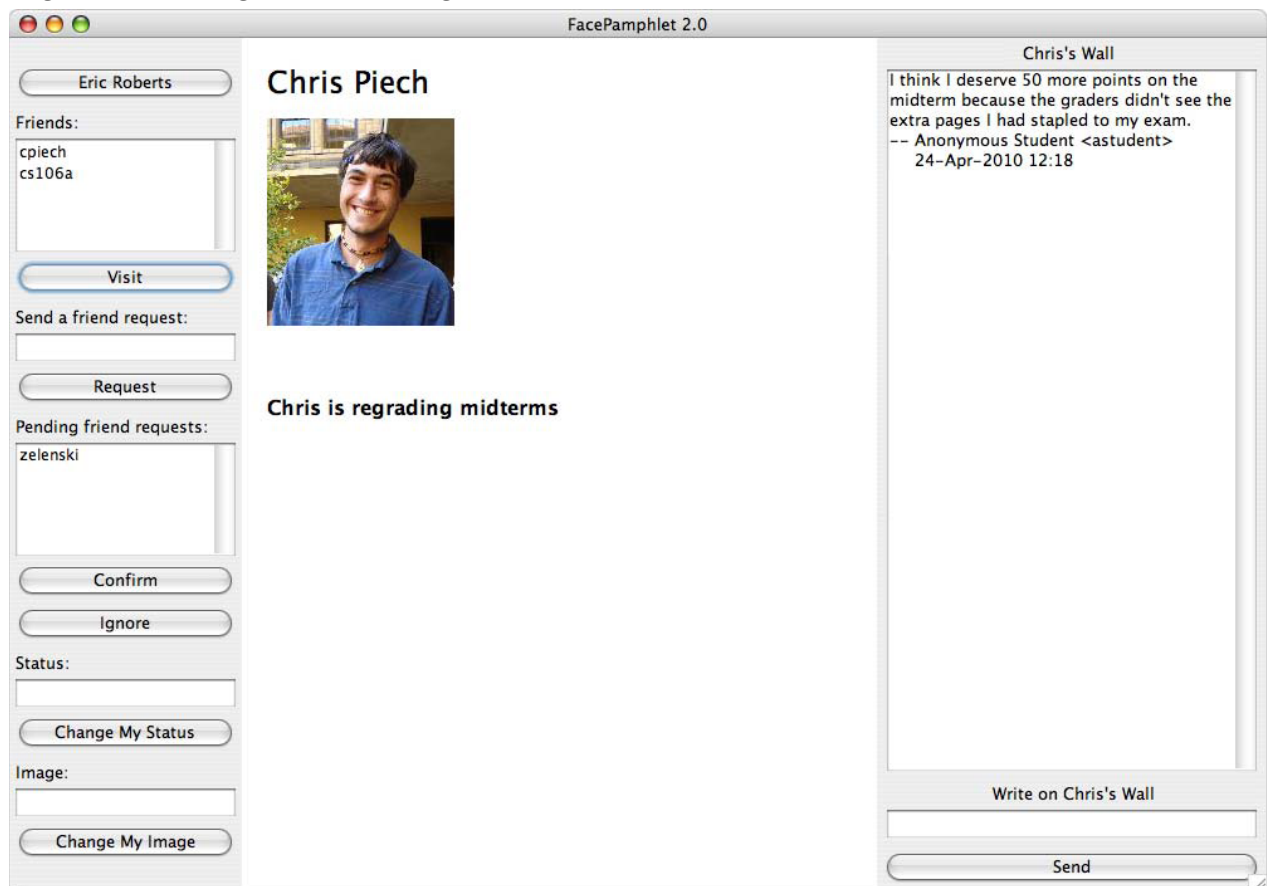
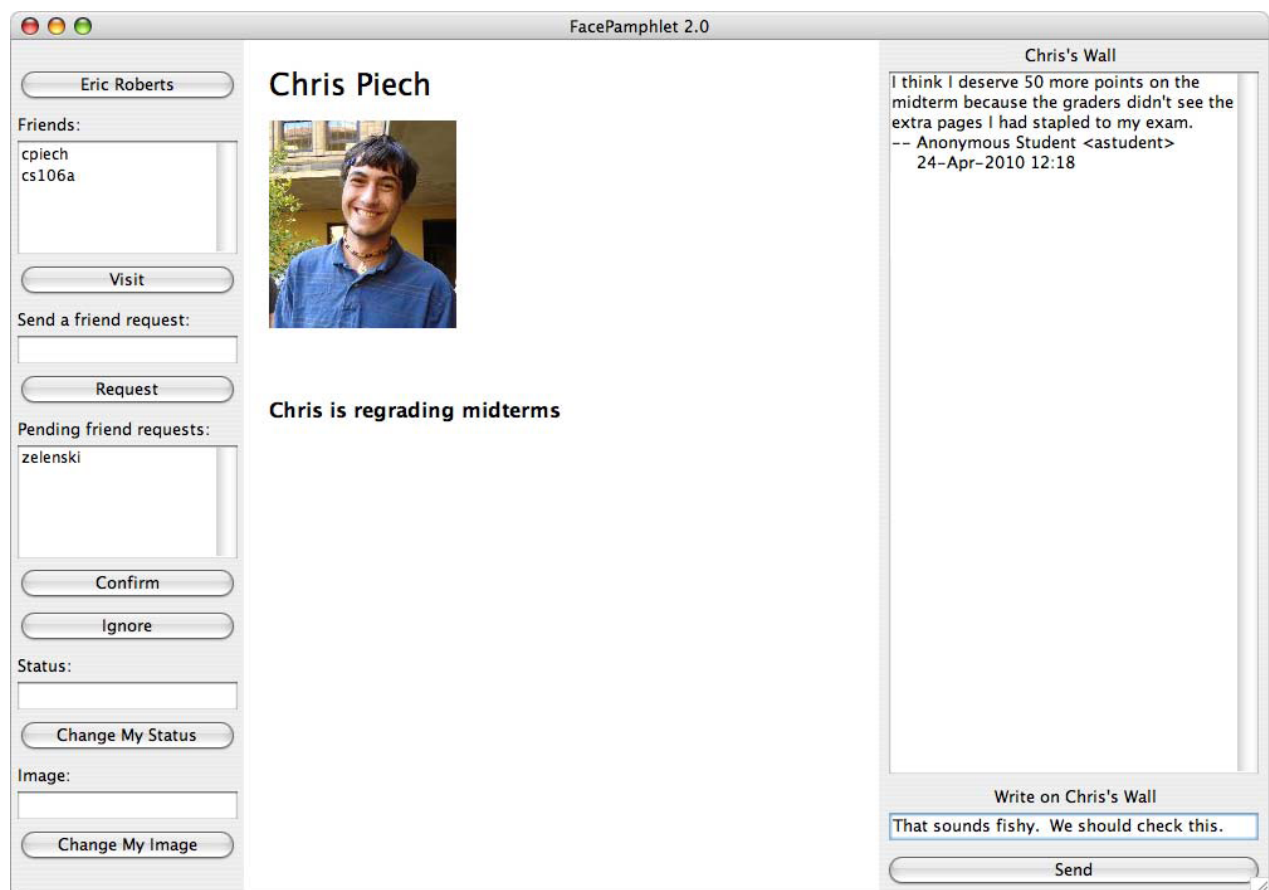


Figure 5: Writing on Someone's Wall (before clicking Send)



Now that I'm visiting Chris's page, I can post a message on Chris's wall. To do so, I simply type a message into the message field at the bottom of the window, and click the **Send** button. When I do, my message is added to the beginning of Chris's wall (as in the real Facebook, it makes sense for more recent posts to appear first). Clicking **Send** also clears the text field so that it is ready for another message. The situation before and after clicking **Send** is illustrated in Figures 5 and 6.

The client-server model

One of the most exciting things about FacePamphlet 2.0 is that it communicates over the network, making it possible for users to interact, in much the same way that they do on social networking sites. 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 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 on the server machine.

Figure 6: Writing on Someone's Wall (after clicking Send)

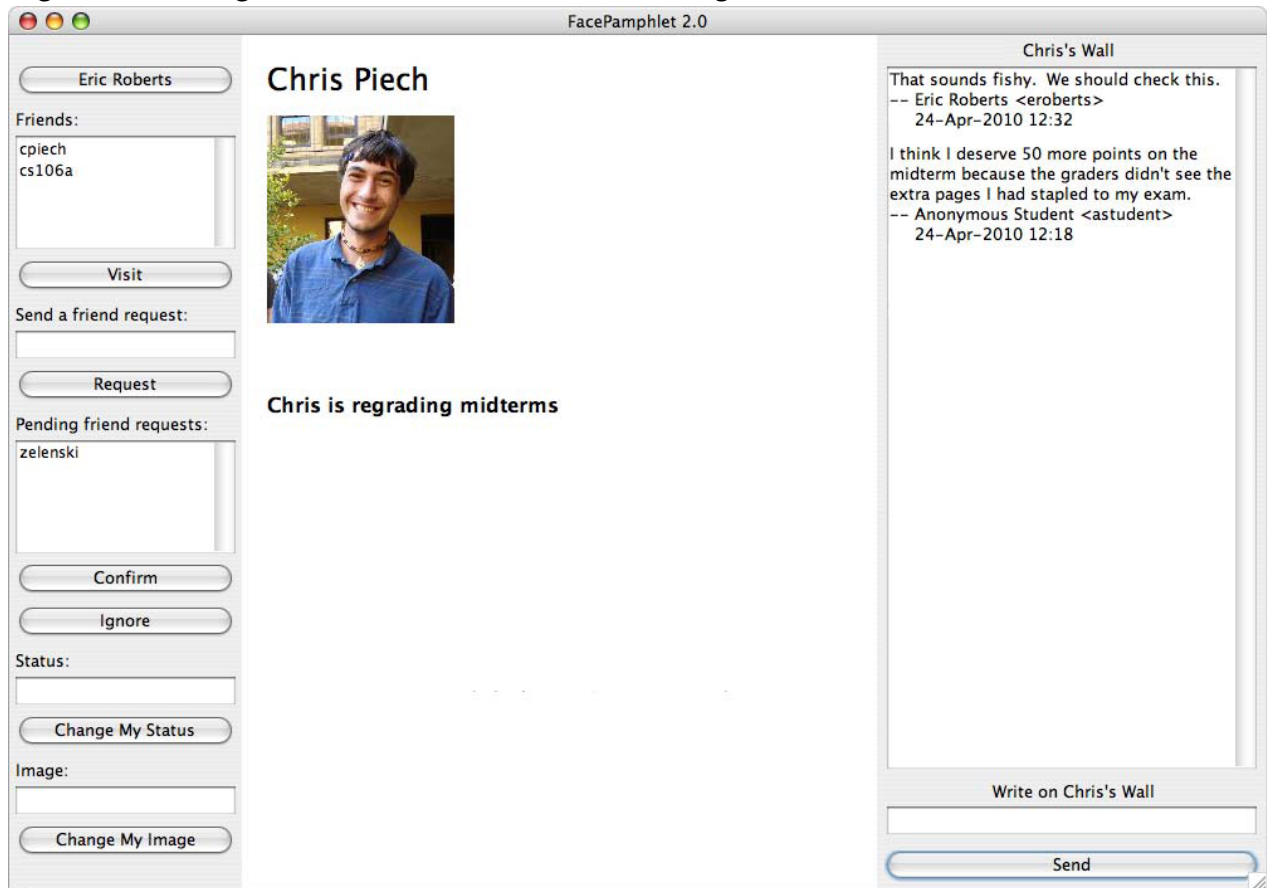
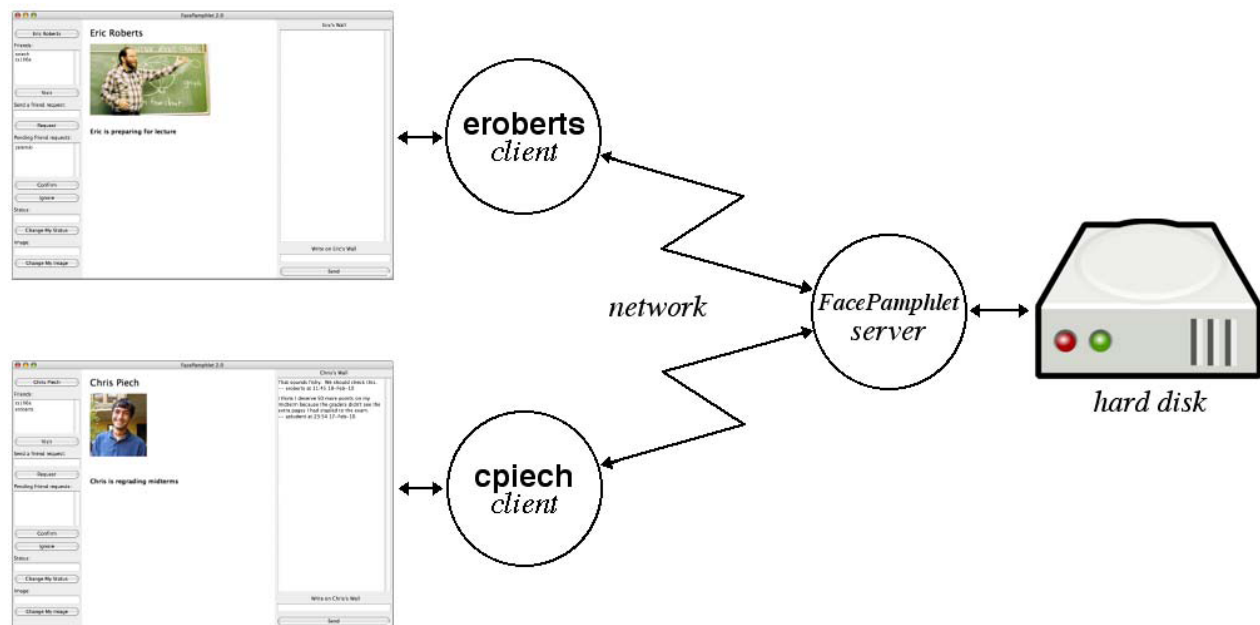


Figure 7: The client-server architecture of FacePamphlet



The contents of the starter project

Given all the things it does, this assignment sounds huge. Fortunately, 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. The only class you need to implement yourself is **FacePamphlet** itself. The version of **FacePamphlet.java** in the starter project includes a tiny bit of code to get you started, but most of that class is yours to write.

In addition to the stub version of **FacePamphlet.java**, the starter project includes the library archive **facepamphlet.jar**, which contains a precompiled collection of classes that you will need to implement the FacePamphlet application, as follows:

- **FPConstants**—This class defines a set of constants for you to use in the main program and therefore plays much the same role as the **YahtzeeConstants** in Assignment #4. You need to be familiar with the constants this class exports and use those names instead of their values in your code.
- **FPRepository**—This interface defines the notion of a *repository*, which is the part of the FacePamphlet application responsible for storing information about the various users who are part of the FacePamphlet network.
- **FPLocalRepository**—This class implements the **FPRepository** interface using the local file system and does not try to share information across the network. This is the class you should use during the debugging phase.
- **FPNetworkRepository**—This class implements the **FPRepository** interface using the client-server model, thereby allowing different users to share information.
- **FPScrollableList**—This class implements a scrollable list. Two instances of this class appear in the user interface, both on the west side panel. The first shows the list of friends, and the second shows the pending friend requests.
- **FPScrollableTextArea**—This class implements a scrollable text area. The only instance of this class in FacePamphlet is the “wall” area on the east side panel.
- **FPTools**—This class exports two static methods that make it possible to convert back and forth between images and strings.

This handout does not contain a detailed description of these classes. To find out how to use these classes, you need to consult the **javadoc** documentation pages on the CS106A web site. To get you started, Figure 8 shows the beginning of the **javadoc** for the **FPRepository** interface. Because understanding how the repository works is essential to making progress on this assignment, the following section describes its structure in more detail.

Figure 8: The javadoc page for the **FPRepository** class

stanford.facepamphlet

Interface FPRepository

Object

└ `stanford.facepamphlet.FPRepository`

All Known Implementing Classes:

[FPLocalRepository](#), [FPNetworkRepository](#)

```
public interface FPRepository
```

This interface defines the methods in a *repository*, which stores user data for FacePamphlet. The library archive for the FacePamphlet assignment defines two concrete classes that implement this interface, as follows:

1. The `FPLocalRepository` class, which uses the file system to create a repository on your own machine. As a result of this design, the data you enter into the repository will be persistent (in the sense that it does not go away when the program exits) but not shared with other users. You should test your code with the local version of the repository to make sure you have everything working.
2. The `FPNetworkRepository` class, which implements the repository using a client-server model that runs over the network. This implementation allows users on different machines to share information.

The best way to think about a `FPRepository` object is to regard it as a map that associates property names with values. Each property in the repository is identified by a string in the form *id.key* where *id* is the id of the user whose values you are looking for, and *key* indicates the name of the specific property you want to find. For example, if you want to find the full name of the user `eroberts`, you would write

```
String fullName = repository.getProperty("eroberts.name");
```

In all likelihood, the user id will be stored in a variable, so that the actual code will presumably look more like this:

```
String fullName = repository.getProperty(homeId + ".name");
```

A repository, however, is more sophisticated than a map. In addition to getting and setting properties, the repository supports the following operations that are more closely tied to the FacePamphlet application:

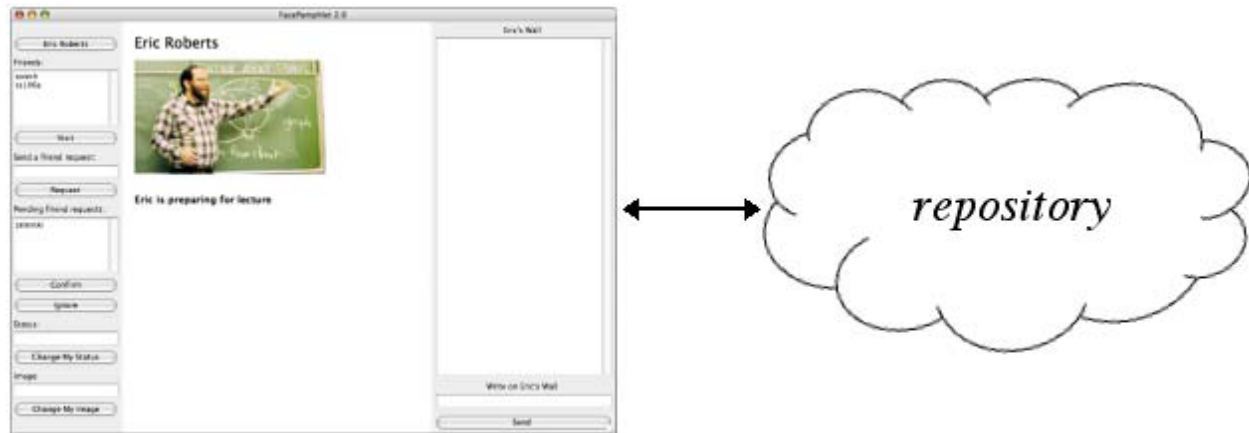
- *Friend management.* In a repository, the value of a property you own is readable only by users that you designate as *friends*. The interface therefore includes methods for requesting, confirming, and ignoring friends.
- *Writing on walls.* The repository allows you to write on the walls of your friends. Messages posted on a wall are prepended to earlier messages so that the most recent messages appear first.

Method Summary

void	close()	Closes the repository.
void	confirmFriendRequest(String id)	Confirms a friend request from the user with the specified id.
String[]	getFriendRequests()	Returns an array containing the ids of the users who have requested to be your friends.
String[]	getMyFriends()	Returns an array containing the ids of the users who are currently your friends.
String	getHomeUserId()	Returns the id for the user who is logged in to the repository.
String	getProperty(String property)	Gets the value associated with the specified property, which is indicated as a string in the form <i>id.key</i> .
String[]	getUsers()	Returns an array all user ids currently in the repository.
void	ignoreFriendRequest(String id)	Ignores a friend request from the user with the specified id.
boolean	isMyFriend(String id)	Returns true if the specified id is a friend, and false otherwise.
void	requestFriend(String id)	Extends a friend request to the specified person.
void	setProperty(String property, String value)	Gets the value associated with the specified property, which is indicated as a string in the form <i>id.key</i> .
void	writeWallMessage(String msg, String id)	Adds a message to the wall for the specified id.

The FacePamphlet repository

Although Java does a reasonably good job of hiding the complexity involved in network communication, the details of the client-server implementation are beyond the scope of CS106A. To make this assignment manageable, we've written the code for the server, along with the network-handling side of the client. The existence of this code allows you to simplify your conceptual model of the FacePamphlet application so that it looks like the following diagram:



This diagram is similar to the one in Figure 7 except that the details of the client-server connection have been replaced by the more nebulous concept of a repository, which is an abstract name for that part of the application responsible for storing and retrieving information about the users in the FacePamphlet community. Understanding how the repository implements those operations is not essential to you as you create the user interface; those details are, in some sense, hidden behind the cloud. What you do need to understand is what capabilities the repository provides.

In Java, one of the standard ways to specify the behavior of an object without revealing the details of its implementation is to define an *interface*, which specifies a set of methods that are then implemented by one or more classes. The interface itself supplies only the header lines of the methods exported by the interface; the actual code for those methods is supplied by each of the classes that implement it.

As noted in the preceding section, the **facepamphlet.jar** file supplied with the starter project includes a **FPRepository** interface, along with two classes that implement that interface. The **FPLocalRepository** class stores its data using the file system of the computer you're using. The **FPNetworkRepository** class, by contrast, stores its data on a server, where it can be shared among the users in the FacePamphlet community.

Given that so much of the excitement associated with the FacePamphlet assignment comes from sharing data with other users, you might be tempted to use the **FPNetworkRepository** class right from the beginning. That approach, however, would be a serious mistake. Code that uses the network is more difficult to debug than code that runs entirely on your local machine. Worse still, running a buggy FacePamphlet application in a network setting can interfere with other users and might even crash the server. The best strategy—which is described in more detail in the discussion of milestones

later in this handout—is therefore to debug your program using the **FPLocalRepository** class and to change to the **FPNetworkRepository** class only when you get everything working. Making that change, however, is easy because the two classes implement the same interface.

To get a better idea of how the **FPRepository** interface works, it helps to look at the **javadoc** shown in Figure 8. As the comments at the beginning of the interface description indicate, it is useful to think about a repository as a map that associates property names with values. In contrast to a traditional map, however, properties in a repository require two levels of specification. The repository contains information for many different users and, for each user, must keep track of several different pieces of information. Property names therefore consist of a string in the form **id.key**, where **id** indicates the user id and **key** indicates the specific information you want for that user. A list of the keys used in the FacePamphlet application appears in Figure 9. By default, a property can be written only by the owner and read only by the owner or one of the owner’s friends.

In most cases, you will need to construct the property name by concatenating the user id and the key value, separated by a period. The **javadoc** comments for the **FPRepository** interface illustrate the use of this technique.

Figure 9: Predefined keys used by the repository

name	The full name of the user. This key can be read by anyone, not just friends.
password	The user’s password. Only the owner can read this particular key.
image	A string representation of the user’s profile image. When you read or write this key, you need to use the methods in FPTools to convert back and forth between GImages and strings.
status	The user’s current status, which is the string that appears under the image in the profile.
friends	A list of the users friends, one per line. This key cannot be written directly by the owner because doing so would circumvent the rule requiring both parties to agree to a friendship. The friends key can, however, be read by any user.
requests	A list of the pending friend requests, one per line.
wall	The contents of the user’s wall, which typically consists of multiple lines.

Implementation milestones

Even though the classes in **facepamphlet.jar** implement many of the more advanced features of this assignment, there is still quite a bit of code you have to write. As always, writing that code will be much easier if you implement the application in stages, making sure to get each part working before moving on to the next. The rest of this section outlines a series of milestones designed to make this assignment much more manageable.

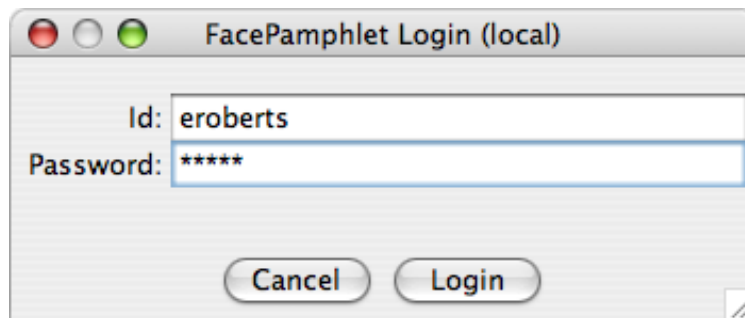
Milestone 0: Run the starter code

Before you write any of your own code, it is useful to familiarize yourself with what you already have in the starter folder. The starter version of **FacePamphlet** looks like this:

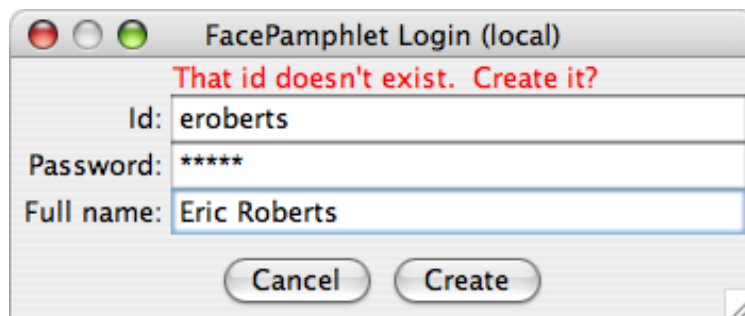
```
public class FacePamphlet extends GraphicsProgram
implements FPConstants {
    public void init() {
        repository = new FPLocalRepository();
        // You fill in the rest
    }

    /* Instance variables */
    private FPRepository repository;
}
```

There's not a lot there, but the code does create an instance of the **FPLocalRepository** class, which is the type of repository you want during the debugging phase. Calling the constructor brings up a dialog box that lets you enter your user id and password (the one we gave you, not your Stanford one). When I run FacePamphlet, I get this dialog:



In the networked version of the repository, the constructor will establish a connection with the server and check the password against the database we've created for the class. In the local version, the constructor asks if you want to create a new user, as follows:



If you type in your full name and click **Create**, you'll be ready to move on to your first real task.

Milestone 1: Create the west side 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. The other interactors are either **JButtons**, **JLabels**, or **JTextField**s, as described in Chapter 10 of the text.

For each of the interactors in the control panel, you have to decide whether you need to store it in an instance variable so that you can use it in other parts of the code. The **JLabels** on this panel, for example, are simply informational text; once you create them, you never need to refer to them again. The **ScrollableList** objects and the **JTextField**s, by contrast, must be stored in instance variables because you need to call their methods. The **JButtons** 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 include the button label as their action command.

The one button that requires special handling is the one at the top of the window. The label on this button is the full name of the home user, which you need to get from the repository. The first step in that process is to get the id of the home user and store it in an instance variable like this:

```
homeId = repository.getHomeUserId()
```

The second step is to get the full name by calling

```
repository.get(homeId + ".name")
```

Your task in the first milestone is simply to add all of the interactors to the west side panel, even though they won'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. At the very least, you should define a method called **initWestPanel** (or something like that) that sets up this side panel; you could, however, break the initialization process down into even smaller pieces, if you can identify logically connected subtasks. Remember that decomposition is your friend.

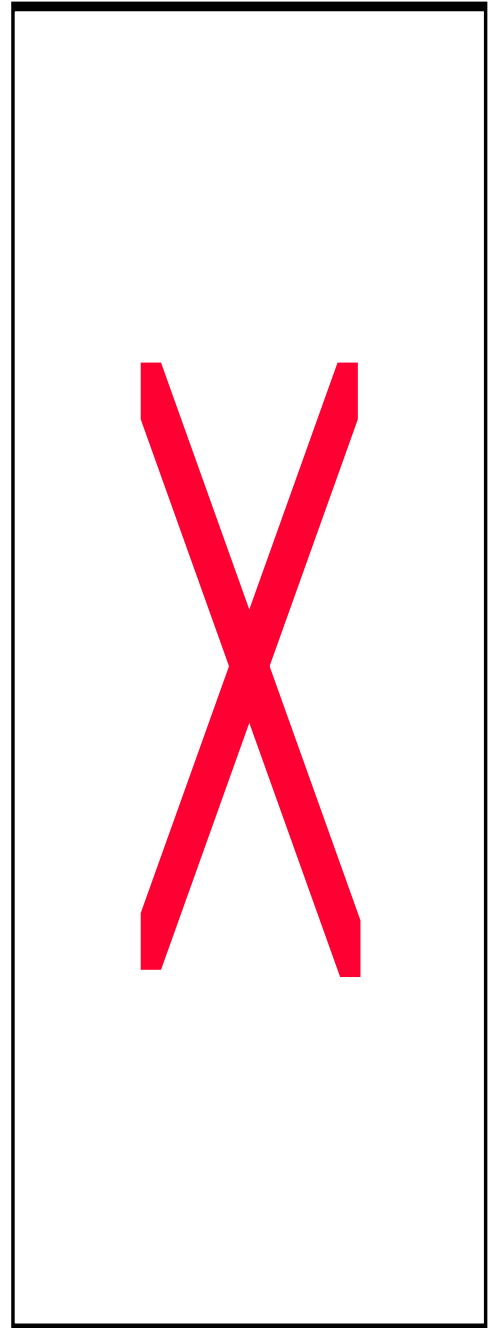
The panel on the east side is in most respects simpler than the one on the west, but there are a few new issues that you have to consider. The most obvious difference is that there is only one large interactor on this side, which is an **FPScrollableTextArea**. That interactor, moreover, needs to take up whatever vertical space is available. Because the side panels use a **TableLayout** manager, you can ensure the correct formatting by adding a third parameter to the **add** method call, as follows:

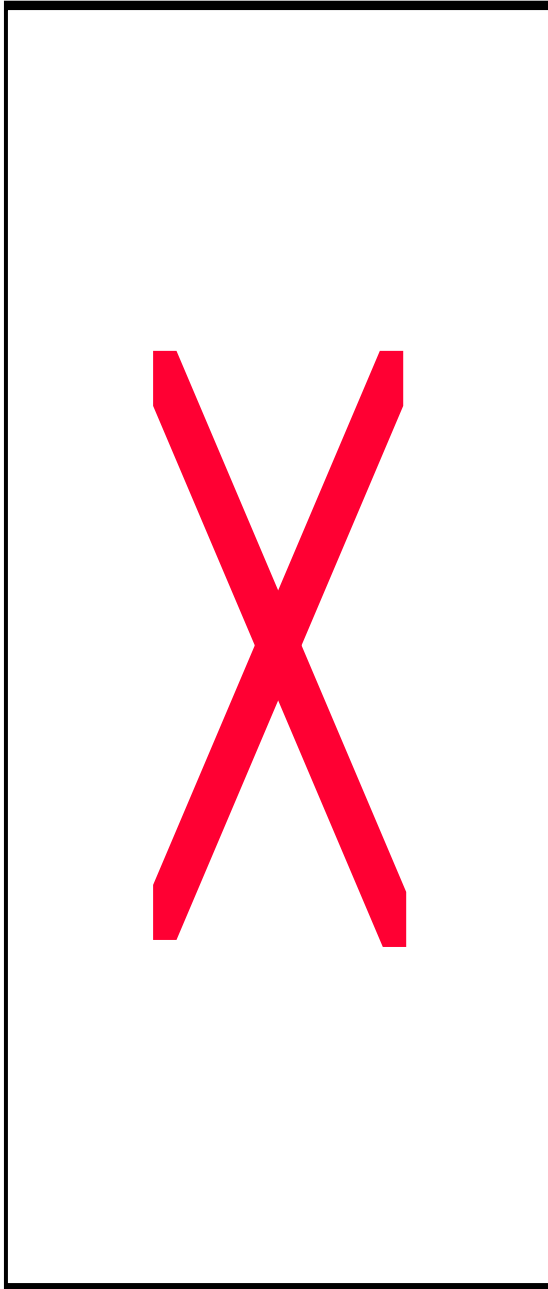
```
add(wallArea, WEST, "weighty=1");
```

The **weighty=1** specification indicates that any available space in the *y* dimension should be assigned to this object. (See page 392 in the text for more discussion of these constraints.) Another important difference is that **JLabels** on the east side panel change depending on whose wall you're visiting. 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**. As noted earlier in this handout, the id of the user whose wall you're viewing is called the *visit user*, and you'll find it convenient to store this id in an instance variable, just as you did with the home user in Milestone 2. Also, given that these labels display the first name of the user, it probably makes sense to write a private helper method that extracts the first name from a full name. You'll need this method again for the graphical display in Milestone 3.

You also need to note that the **JLabels** on this side are centered rather than aligned on the left, which is the default behavior for the **JLabel** class. To center a **JLabel**, you need to set its horizontal alignment to **JLabel.CENTER** by passing that constant as a second argument to the **JLabel** constructor.

Although the text that gets displayed in the wall looks complicated because it contains the date, time, and sender information for each message, updating the contents of the wall is actually the easiest part of this milestone. The repository takes care of formatting the wall messages. All you need to do is get the value of the **wall** key for the visit user and display that string in the **FPScrollableTextArea**.





Milestone 3: Create the center panel

Your next step in implementing the FacePamphlet application is to put together the graphical display that occupies the center of the window, which contains the **GCanvas** that is automatically created as part of any **GraphicsProgram**. As you can see from the example on the left, this display consists of four graphical objects:

- A **GLabel** giving the full name of the visit user
- A **GImage** showing their picture
- A **GLabel** indicating their status
- A **GLabel** reporting a user message

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

For this milestone, your primary task is to arrange the various graphical objects in the window. Given that there are only four of them, this problem is much easier than the Breakout assignment, where you had to create an array of bricks, a paddle, and a ball. Moreover, given that these objects don't move, there is no need for an animation loop.

The values stored in these graphical objects, however, will change. If you reset your status, the contents of the status **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, as described on the following page. Once you start working with the network repository, it's possible for values to change even when you don'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 each field.

When you regenerate the graphical display, you also need to update the contents of those parts of the side panels that might have changed, such as the friends list, the pending request list, and the wall of the visit user. The repository has methods for getting the appropriate 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. In terms of decomposition, it makes sense to have a general **update** method that then calls subsidiary methods to update the east, west, and center panels. When you get around to responding to actions in Milestone 4, you can call the **update** method after every user action. For now, you can simply add a call to **update** at the end of the **init** method.

Of the graphical objects shown in the center panel, two are quite straightforward: the name at the top and the status shown in the middle of the page. To update the name, all you need to do is retrieve the full name from the repository and then adjust the font and location as specified by the parameters in **FPConstants**. The status label is pretty much the same, particularly given that you are not required to ensure that the status message fits inside the window.

The hardest part of this milestone is displaying the image, which is complicated for two reasons:

- You must convert images to strings before storing them in the repository. The **FPRepository** interface assumes that all values stored in the repository are strings. Thus, if you want to store an image in the repository, you need to convert it to a string form before storing it, and then convert the string back into an image before you display it on the screen. To do so, you need to use the methods in the **FPTools** class that perform this conversion. The details for doing so are provided in the **javadoc** for the **FPTools** class.
- You must check to see if the image fits in the available space and adjust its size if necessary. One of the requirements for this assignment is that the image fit within an area of the graphics windows whose dimensions are defined by constants in the **FPConstants** interface and by the width of the graphics window. If the image fits inside that area, no scaling is required. If not, you need to use the **scale** method in the **GImage** class to reduce the size of the image until it stays within the boundaries. Moreover, to ensure that the images are not distorted, that scaling must be **uniform**, which means that the same scale factor must be applied in both the x and y dimensions. Thus, you need to choose a single scale factor that ensures that the resulting image will fit within both the horizontal and vertical bounds.

Displaying the information message at the bottom of the screen can also be a bit tricky. All the other information in this window comes from the repository. By contrast, the text that gets displayed on this line comes from two sources: messages that show that a particular request has succeeded (which are often called **acknowledgments**) and error messages.

In this assignment, you are required to generate an acknowledgment when the user clicks the **Request**, **Confirm**, or **Ignore** button. The precise format of the acknowledgment is up to you, but each one should include the full name of the user involved, as in the line

You are now friends with Chris Piech.

that appears in Figure 3.

Reporting error messages is a little more complicated. In Java—as in most modern languages—the usual strategy for reporting errors and other unusual conditions is to use exceptions. The methods in the classes that implement **FPRepository** use this strategy and throw an **ErrorException** whenever an error condition occurs. To display error messages in the center panel, you need to catch those exceptions, extract the text of the error string by calling **getMessage** on the exception object, and then display that string in the **JLabel** at the bottom of the window. In pseudo-code form, the structure you need looks like this:

```
try {
    Code in which errors might occur.
} catch (ErrorException ex) {
    String errorMessage = ex.getMessage();
    Code to report the error on the screen.
}
```

Exceptions and the syntax of the **try/catch** statement are described in the text beginning on page 490.

Debugging Milestone 3 will require some discipline and creativity on your part. The problem you need to overcome is that you won't actually generate the button clicks or error messages until you get to Milestone 4. Although it's tempting simply to rush ahead to the next milestone and do all your debugging there, it will probably save you time in the long run if you take the time to test that all the graphical features of the center panel are working before you try to implement the interactive features of the assignment.

One strategy you might try is to write test code that sets the values of your own properties and then see whether your **update** method displays those values correctly. For example, you might use the following method to test your status display:

```
private void testStatusDisplay() {
    repository.setStatus(homeId + ".status", "testing status");
    update();
}
```

If the screen correctly reports the message

Eric is testing status

the portion of your code that displays status messages appears to be working. Similarly, you can test whether informational messages work by explicitly calling the code that

displays them. If you get that working now, it will be easier to find problems in your code for the later milestones.

You can also test your code for Milestone 3 by visiting the FacePamphlet page that belongs to the virtual **cs106a** user, which is defined in both the local and networked versions of the repository. Each new user is automatically signed up to be a friend of **cs106a**, which means you can read the values of the fields belonging to that user even before you have your friendship code working. If you explicitly set the visit user to the string "**cs106a**" and call your **update** method, you should see the page for the **cs106a** user. Both implementations of the repository implement image and status information for this user, so you can test your code more easily.

Milestone 4: Implement the interactors

At this point in the program, you have a graphical user interface with various buttons and fields on the two side panels. The only problem is that none of these interactors actually does anything. To make your application do all the exciting things it's supposed to, you need to complete two tasks:

- Add initialization code to ensure that the program listens for action events generated by the buttons and other interactors in the side panels.
- Define an **actionPerformed** method that implements the action. This method should look at the event object to determine what action the user has requested and then call a separate method to implement that action.

Adding the action listeners is reasonably straightforward, particularly for the buttons. To add the program as an action listener for every button, all you need to do is call

```
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 **friendsList**, you can set things up so that double-clicking on a name in the list does the same thing as clicking on the **Visit** button. All you need to do is add the following statements to your initialization code:

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

The more interesting part Milestone 4 is writing the code for **actionPerformed**. This method needs to look at the action command in the event and determine what button was activated by looking at the action command. Your implementation then needs to take some response to that action. The code for **actionPerformed** is therefore likely to be structured as a cascading series of **if** statements that checks for a particular action command and, if the action matches, calls a method to implement it, passing along any additional information that action needs as arguments. For example, the code in **actionPerformed** that responds to the **Change My Status** button will look like this:

```

public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Change My Status")) {
        changeMyStatus(statusField.getText());
    } else if . . .
}

```

The **changeMyStatus** method (which you still have to write) takes a string argument and uses that value to update the **status** key for the home user.

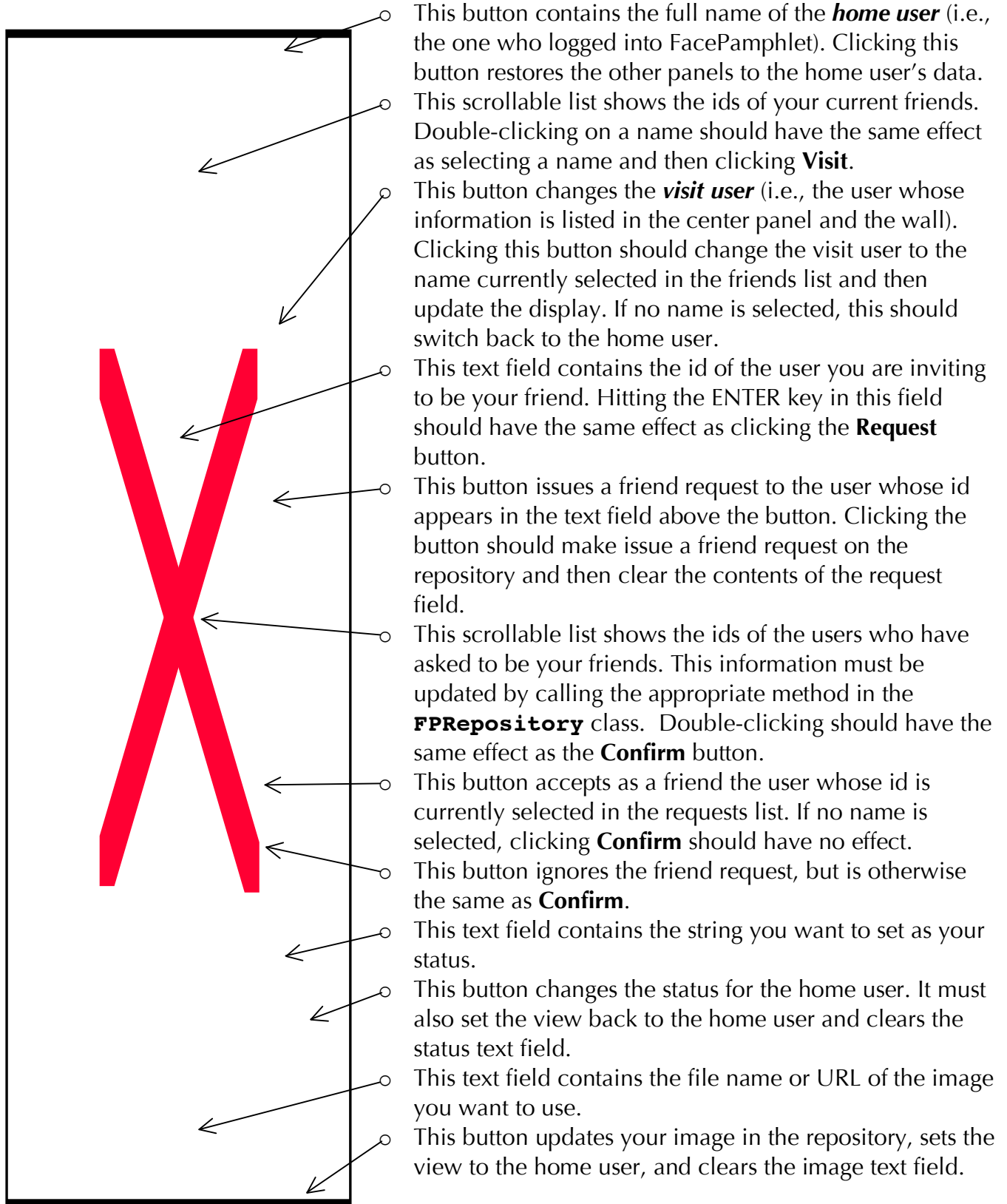
Implementing this milestone will be easier if you implement and test the interactors one at a time. You could start, for example, with the **Change My Status** button, make sure that works, and only then move on to the other buttons in that panel. In all likelihood, the code you write for Milestone 4 will reveal bugs in the code you wrote for the earlier milestones but had yet to discover. If you try to get everything working all at once, it will be much harder to isolate the source of the problem and direct your energies at that point.

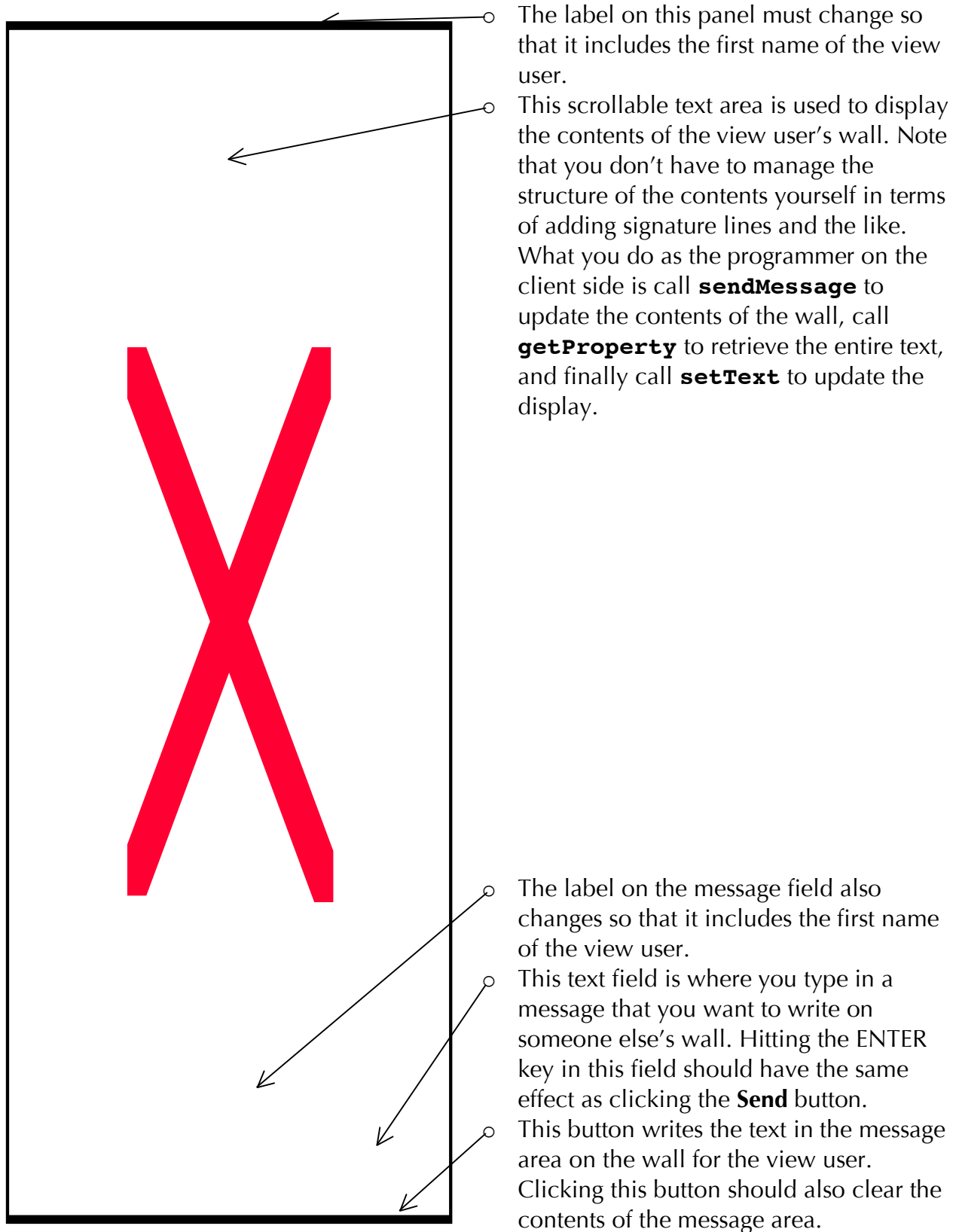
As you implement the action-handling methods for the various interactors, you should use Figures 10 and 11 as a checklist to make sure that your code implements the actions correctly and takes account of the special cases that are required by this assignment.

As you write the code for these interactors, there are a couple of issues that you need to keep in mind:

- The contents of the text field associated with the **Change My Image** button should be the name of a file in the **images** folder for the project or the URL for an image somewhere on the network. Either of these strings can be passed as the argument to the **GImage** constructor, which will create the corresponding **GImage** object. As described in the notes for Milestone 3, you need to use the methods in the **FPTools** class to convert the image to a string before you send it to the repository.
- Your code that responds to actions must correctly display acknowledgments and error messages at the bottom of the center panel. Thus, the code for **actionPerformed** will need to catch error exceptions and then ensure that the error message gets stored somewhere in which your **update** method can find it.
- The easiest way to debug your code is to create several users in your local repository and make sure that the users can interact among themselves. You can even have several copies of the repository running at the same time, each of which can be logged in as a different user. You could then issue a friend request from one user to the other and then switch users and confirm the friend request. Remember that you may need to click on the top left button to update the display before you can see changes made by another copy of the application.

Figure 10: West Side Panel Action Requirements





Milestone 5: Shift over to the networked repository

If you get everything working with the local 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 an **FPNetworkRepository** object rather than the local version. Assuming that works, you can put up a picture of yourself, keep your status current, invite and confirm (or ignore) invitations from friends, and start interacting in our local version of the Facebook world!

Suggestions for extensions

There are many features that you could add to FacePamphlet that would make it more exciting. Here are a few suggestions:

- Pick your favorite feature from the real Facebook application and implement it for FacePamphlet. As you do, feel free to add new keys to the FacePamphlet repository to keep track of any additional information you need. If you define a new key name, the standard rule is that only the home user can change that information, and only the home user and friends of that user can read it. If you want to define a key that all users can read (as is the case for the **name** and **friends** keys in the predefined list), use a key name that begins with a dollar sign. For example, if you define the key **interests** and store information under that key, all FacePamphlet users will be able to read it.
- As the FacePamphlet application currently stands, the status field in the center panel can easily extend beyond the end of the window. To fix this problem, you can extend your code so that it breaks an overly long string into multiple **GLabels** that are then displayed on successive lines. This strategy is called *line wrapping*. The **FPScrollableTextArea** class already implements line wrapping (as indeed it is by most Java interactors that display multiple lines). That feature, however, is missing from the **acm.graphics** package, so you need to implement it yourself.
- The center panel uses the **acm.graphics** package only because you are familiar with it from earlier assignments. If you want to learn more about the Swing toolkit, you can re-implement the center panel as a **JPanel** instead of using the **GCanvas** supplied by the **GraphicsProgram** class. If you do so, you should change **FacePamphlet.java** so that it extends **Program** instead.
- If you read the detailed descriptions of the built-in keys shown in Figure 9, you will discover that the **friends** key is readable by anyone. The reason for this somewhat unusual behavior—which you probably wouldn't want on your real Facebook page—is that making the **friends** key readable allows you to determine the structure of the entire social network that FacePamphlet supports. That capability, in turn, makes it possible to implement features like Facebook's "friend finder" that suggests new friends based on the number of friends two users have in common.