CSSE 490 – Model-Based Software Engineering

Project Milestone 1

This first Milestone develops the first iteration of the system, "FacePamphlet," that we want to develop. Ultimately we will use this as the basis for developing a Model-Based Software Engineering environment to expand upon this system.

We borrowed this assignment from Professor Mehran Sihami at Stanford University, as it is relevant to the type of system that might emerge as a domain. While it is small, it offers potential for evolution. Hence, it is relevant to our Model-Based Software Engineering venue.

Objective

To build an initial version of the FacePamphlet application in Milestone 1.

1) Understand the assigned system.
2) Design the FacePamplet application being mindful of the models.
3) Develop Iteration 1 of the FacePamplet application.
4) Demonstrate that it works and meets the specification.

Due Date

11:59 p.m., Friday, Week 2, (March 18th, 2011).

Tasks

This is a summary of tasks that are detailed in the pages that follow.

1. Assemble the GUI interactors.
2. Implement the FacePamphletProfile class.
3. Implement the FacePamphletDatabase class.
4. Implement functionality for Change Status, Change Picture, and Add Friend buttons.
5. Implement the FacePamphletCanvas class and complete the implementation of the FacePamphlet class.
6. Implement Loading and Saving social networks from a file.
7. Demonstrate your software for this iteration at your project lab on Friday of 2nd week. Please make sure your instructor is able to access your code.

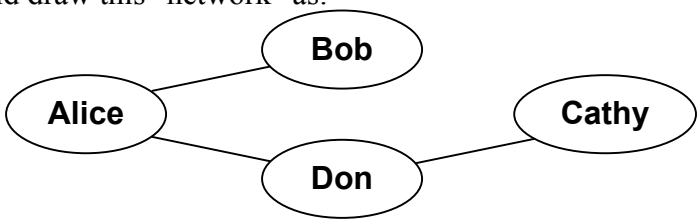
Submitting Your Work

Please submit your Milestone 1 document with your Logical Architecture, Interaction Diagrams, Design Class Diagrams, and source code as a single document to the appropriate Angel dropbox. Please submit a **pdf** file with a cover page containing your Names, Assignment Title, Date, and Campus Mail number. Please name the document: **MS1-FacePamphlet.pdf**.

Milestone 1 Assignment — FacePamphlet

For years, computers have been used as a ubiquitous platform for communication. While email is still the most common medium for computer-based interaction, *social networking* applications (such as Facebook, Orkut, and MySpace[1]) have gained more popularity in recent years. Hence, for the first homework assignment, your team is to create an application that keeps track of a simple social network.

A social network, in the simplest sense, is a means of keeping track of a set of people (each of whom have a "profile" in the social network) and the relationships (usually involving friendship) between them. For example, let's consider a simple social network that contains four people's profiles: Alice, Bob, and Cathy, and Don. Say now that Alice is friends with both Bob and Don (in which case, we consider Bob and Don to automatically be friends of Alice, reciprocally). And Cathy is also a friend of Don. Graphically, we could draw this "network" as:



Here, each profile in the network is represented by a circle containing the name of the profile (more formally, such circles would be called "nodes") and a friendship relationship between two people (which, for our purposes, is always reciprocal) is shown as a line connecting two profiles of people who are considered friends.

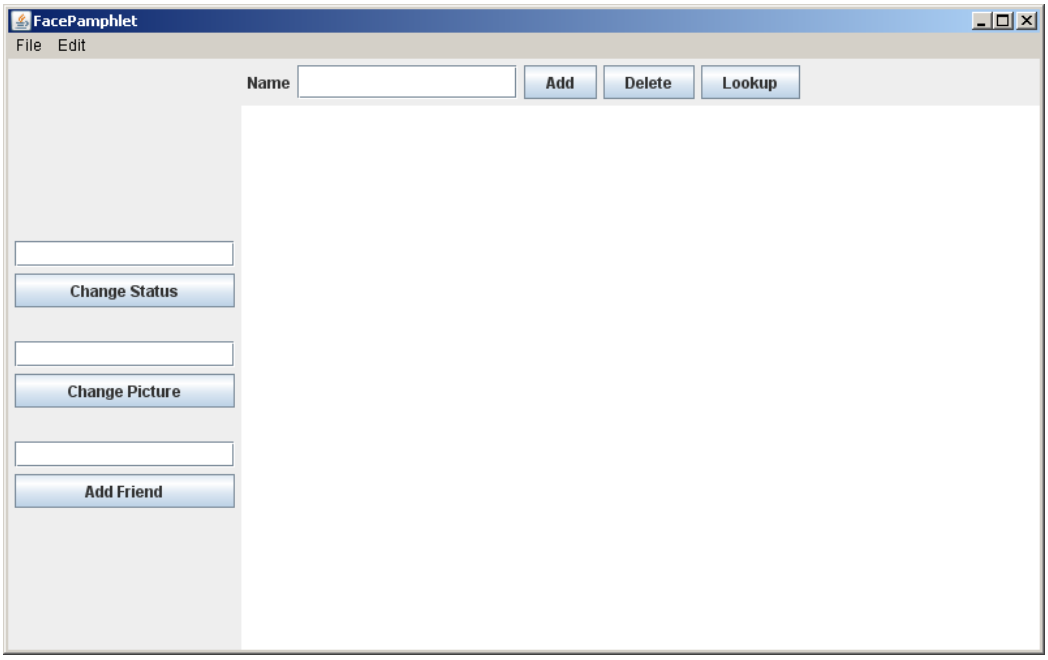
The Assignment

For this assignment, you will create an application that keeps tracks of the information in a simple social network. More specifically, your application will allow for user profiles to be added to, deleted from, or looked-up in the social network. Moreover, for each profile, you will keep track of the person's name associated with that profile, an optional image that the person may wish to display with his/her profile, an optional "current status" for the profile (which is basically just a `String` indicating what activity the owner of that profile is currently engaged in), and a list of friends for each profile.

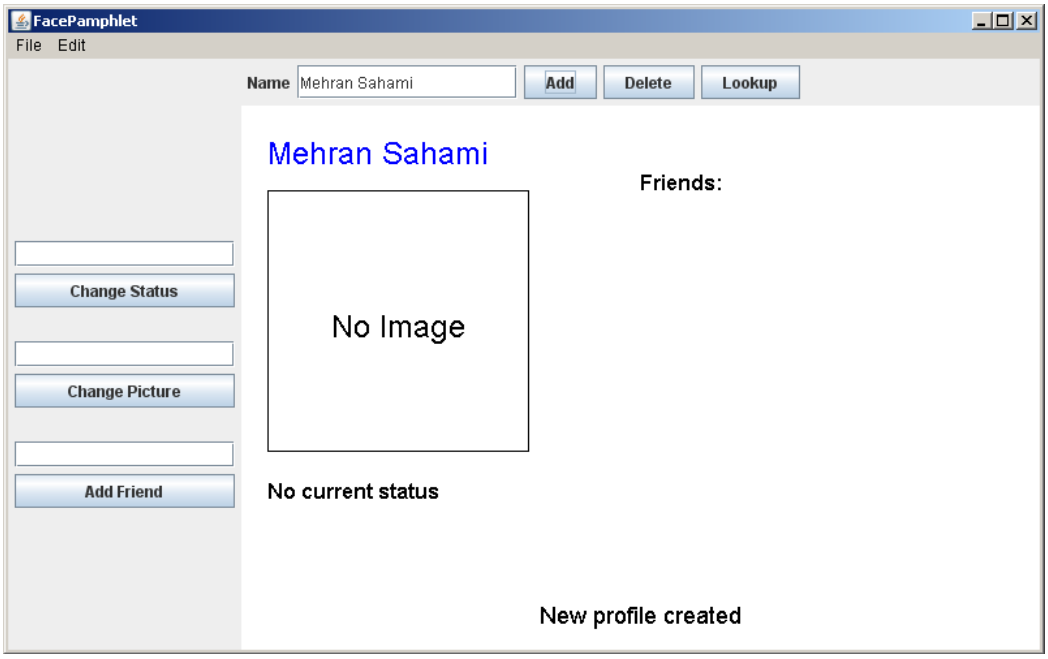
The Program

To see how the program works, we give an example of using the program to create a small social network. Initially, the social network starts out empty (i.e., it contains no profiles). Below we illustrate what the application initially looks like when it is run:

[1] Facebook, Orkut, and MySpace are trademarks of those respective social networking sites. They are referred to here only for eduational expository reasons.



Along the **`NORTH`** border of the application, is a text field entitled **`Name`**, along with three buttons: **`Add`**, **`Delete`**, and **`Lookup`**. To create a new profile, the user would enter a name in the **`Name`** text field and click the **`Add`** button. For example, say we entered **Mehran Sahami** in the text field and clicked **`Add`**. Since there is not already a profile with the name "Mehran Sahami" in the network, the resulting screen would look as follows:



In this profile displayed above, we note five display elements of interest:

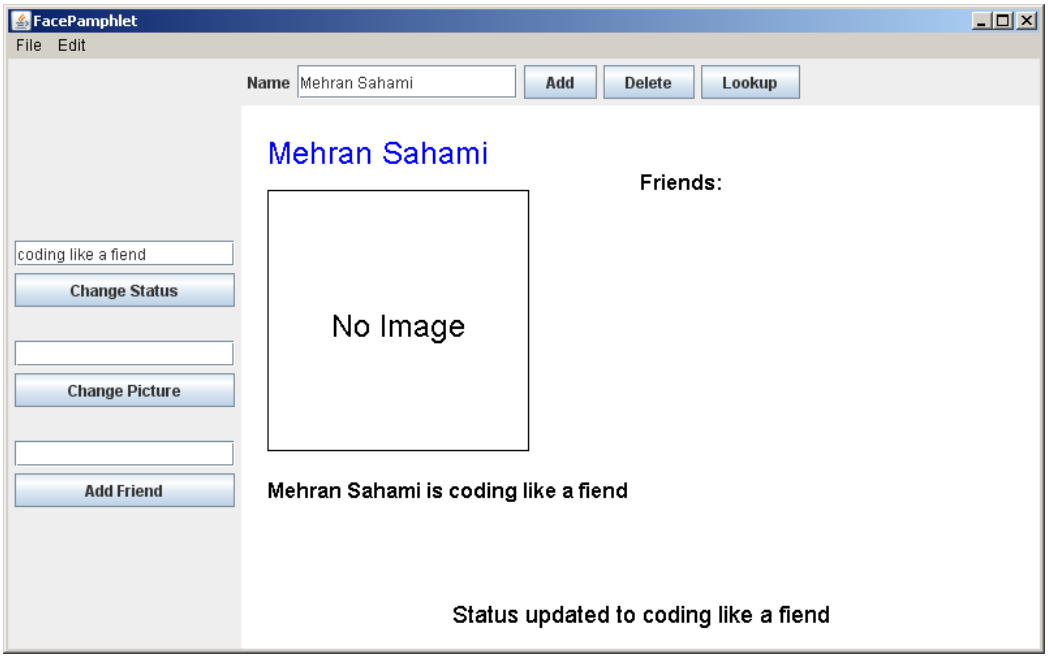
- **Name**: The name associated with the profile ("Mehran Sahami") is displayed prominently in the top left corner of the display canvas. The profile name is

displayed in the color **blue** (though that will not be evident in this black and white handout).

- **Image**: Although there is currently no image associated with this profile, we can see that there is space available to display a picture immediately under the name of the profile.
- **Status**: Under the area for the image, the *current status* of the person with this profile is displayed. Since a newly created profile does not have a status yet set, the display simply shows the text "No current status".
- **Friends**: To the right of the profile's name, there is the header text "Friends:", and space available under this text to list the friends of this profile. Again, since we have just created a new profile, there are no friends yet associated with it, so there are no entries listed under the "Friends:" header.
- **Application Message**: Centered near the bottom of the display canvas is a message from the application ("New profile created") letting us know that a new profile was just created (which is the profile currently being displayed).

Changing Status

Whenever we have a profile displayed in the canvas display area (we refer to this as the *current profile*), the interactors along the `WEST` border of the application can be used to make updates to the current profile. These interactors include text fields and associated buttons to: `Change Status`, `Change Picture`, and `Add Friend`. For example, we can change the status of the current profile above by entering the text **coding like a fiend** in the text field and clicking `Change Status` (or we could simply have pressed the Enter key after typing the in the respective text field). The display updates as follows:



In the screen above we see that the status text associated with the current profile has been changed to the text "Mehran Sahami is coding like a fiend". Moreover, the Application

Message at the bottom of the display canvas has also been changed to reflect the last action taken, namely "Status updated to coding like a fiend".

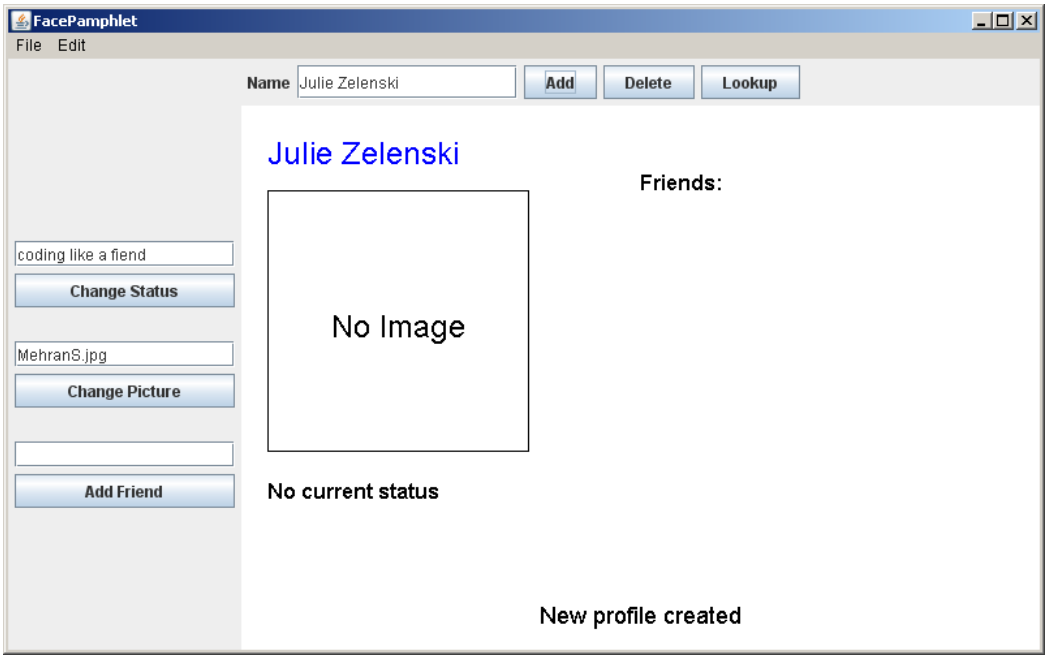
Changing Picture

We can now update the image associated with the current profile by entering the name of a valid image file (in this case, **MehranS.jpg**) in the text field associated with the **`Change Picture`** button and pressing the Enter key (or clicking the **`Change Picture`** button). The display updates as follows:

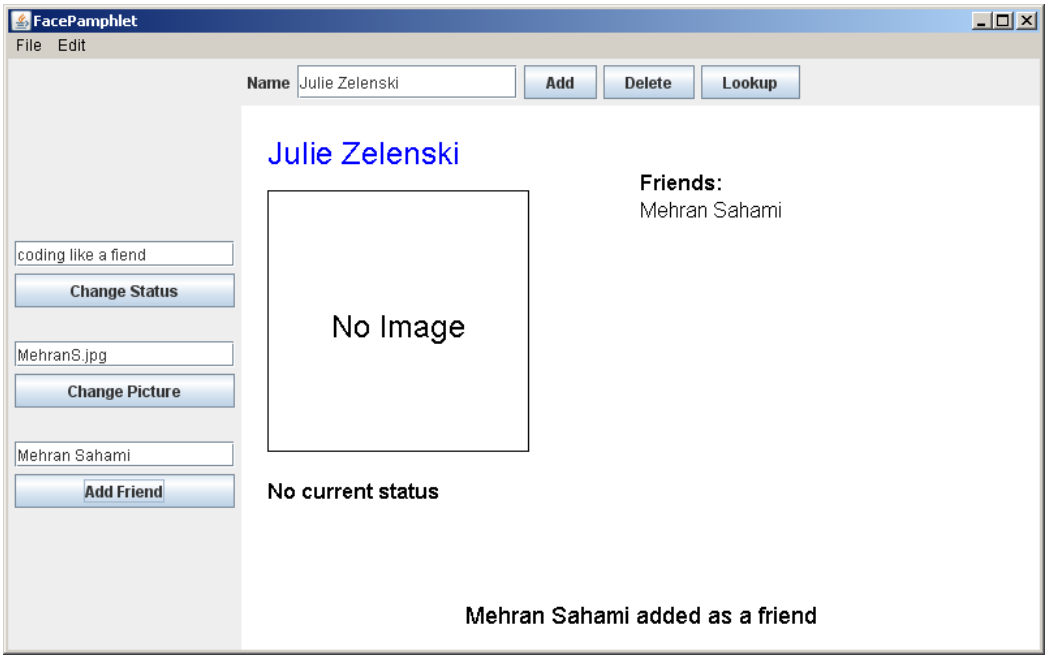
The image area in the current profile now displays (a scaled version of) the image from the file **MehranS.jpg** and the Application Message at the bottom of the display canvas has once again been changed to reflect the last action taken, namely "Picture updated".

Adding Friends

Now, let's add another profile for Julie Zelenski (another intrepid faculty member in the Computer Science department at Stanford) to the social network so that we can show an example of adding a friend to a profile. In the Name text field at the top of the screen, we enter the text **Julie Zelenski** and click **`Add`**. The display now shows the newly created profile (shown on the next page). Note that although a new profile was created for Julie (which has no image associated with it and no current status), the previous values we entered in the text fields for status (**coding like a fiend**) and image (**MehranS.jpg**) are still there simply because the text fields were never cleared. It's important to remember that the values in the text fields do not reflect what is in the current profile we are looking at – rather these fields are simply interactors that allow us to update the values in a profile, and old values entered in these text fields need not be cleared in the program (although this would be a simple extension to add to the program, if you were so inclined).



Since Julie likes to maintain her privacy, she may choose to neither update her image nor her status. But, being the friendly person that she is, she chooses to add Mehran as a friend. This is done by entering the profile name **Mehran Sahami** in the text field immediately above the **`Add Friend`** button and then either clicking the button or pressing the Enter key. After this is done, the display is updated as follows:



In the picture above, we see that **Mehran Sahami** has been added to the list of friends that Julie has, and the Application Message reads "Mehran Sahami added as a friend."

Looking-up Profiles

Recalling that all friendships are reciprocal (i.e., if Julie has Mehran as a friend, then Mehran must also have Julie as a friend), we go to lookup Mehran's profile. This is accomplished by entering **Mehran Sahami** in the Name text field in the **`NORTH`** region of the application and clicking **`Lookup`**. The display then looks as follows:

Here we find that Mehran's profile was updated to have Julie as a friend at the same time that Mehran was added as a friend of Julie in the previous interaction. In this way, the application ensures that all friendships are reciprocal – whenever a friend X is added to a profile Y, then not only is X is added as a friend of Y, but Y should also be added as a friend of X at the same time.

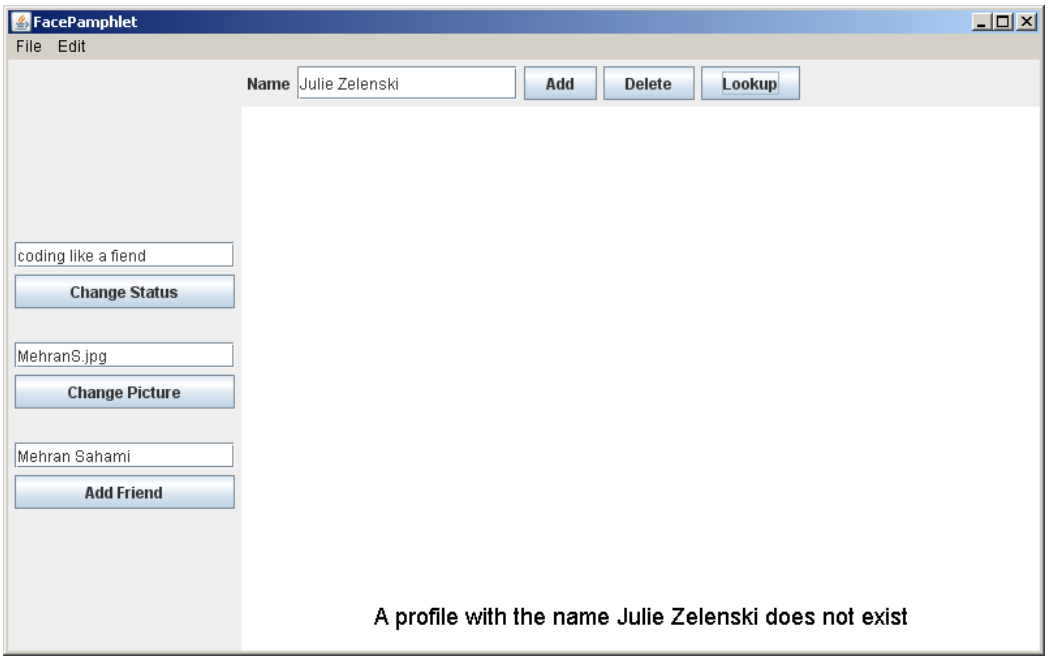
Deleting Profiles

Now let's say that we decide to delete Julie's profile from the social network. We can accomplish this by entering the profile name **Julie Zelenski** in the text field entitled Name (in the **`NORTH`** border region) and clicking the **`Delete`** button. After this is done, the display is updated as shown in the next page. We see in the picture below that after we delete a profile, the current profile being displayed is no longer shown (no matter whose profile that was), and the Application Message simply reports that "Profile of Julie Zelenski deleted".

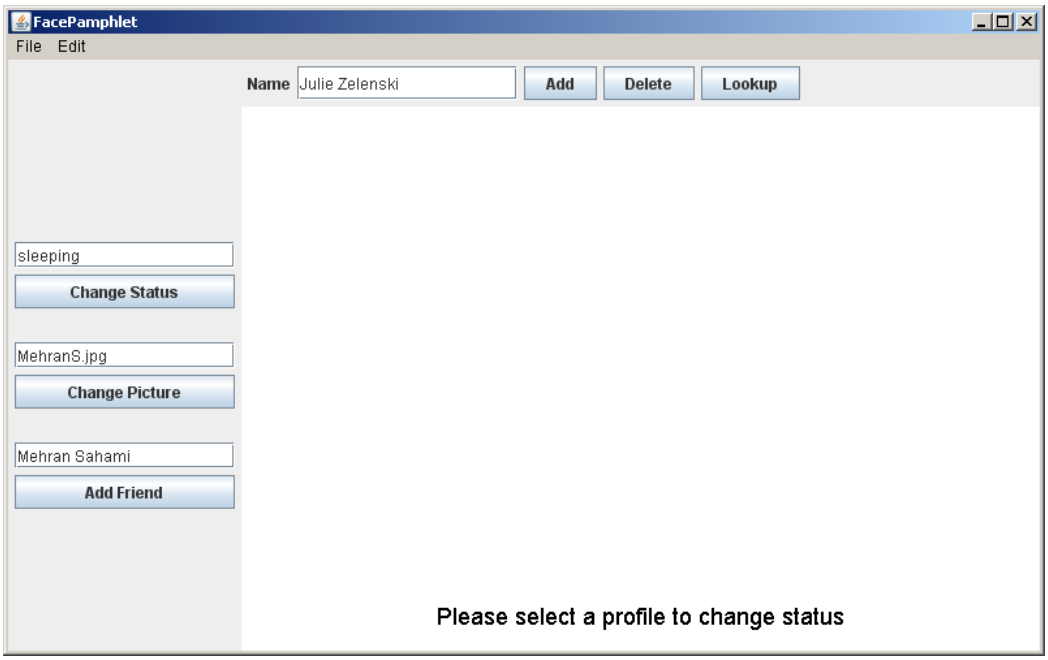
Not only has Julie's profile been removed from the social network, but the profile of *all* members of the social network that had Julie as a friend must also be updated to remove Julie from their list of friends (since it is not possible to be friends with a non-existent profile). So, if we lookup Mehran's profile again by entering **Mehran Sahami** in the text field entitled Name (in the `NORTH` border region) and click the `Lookup` button, the display will look as follows:

Note that Julie is no longer listed as one of Mehran's friends in the display above. She was removed from Mehran's list of friends when her profile was deleted.

To verify that Julie's profile been removed from the social network, we could try to look it up. To do this, we enter **Julie Zelenski** in the text field entitled Name and click the **`Lookup`** button. The display appears as follows:



Note that when we try to lookup Julie's no-longer existent profile, the current profile that was previously displayed is cleared and we are prompted in the Application Message that "A profile with the name Julie Zelenski does not exist". It's important to note that when there is no current profile being displayed (as is the case above), then the interactors in the **`WEST`** border region have no profile to update. Thus, if we were to try to, say, change status by entering the text **sleeping** in the text field and clicking the **`Change Status`** button, the display would update as follows:



As can be seen in the display above, if we try to Change Status when there is no current profile displayed, we are simply prompted with an Application Message saying "Please select a profile to change status". We would receive an analogous prompt (albeit with slightly different wording) if we tried to Change Picture or Add Friend when there was no current profile displayed.

Demo Applet

Although we have described the general functionality of the FacePamphlet program above, there is a web demo applet available on the class web page that will allow you to play with the application yourself and get a better sense for how it works. You can always refer to the workings of that demo applet if you have questions about how particular situations should be handled in your FacePamphlet program. Note that in the web demo applet, there are only two image files that are available. As a result, you will not be able to display any other images using the web demo applet, but these two images should be sufficient for you to still see how the application works.

The Details

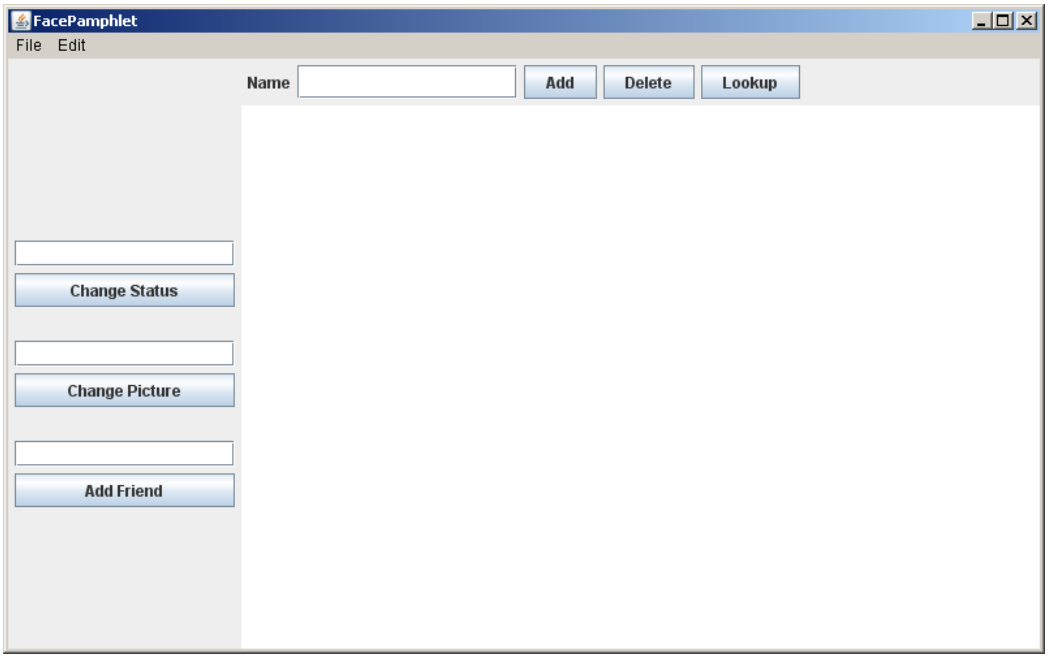
Similar to the NameSurfer assignment, the FacePamphlet program is broken down into several separate class files, as follows:

- **`FacePamphlet`**—This is the main program class that ties together the application. It has the responsibility for creating the other objects and for responding to the interactors in the program.
- **`FacePamphletConstants`**—This interface is provided for you and defines a set of constants that you can use in the rest of the program simply by having your classes implement the **`FacePamphletConstants`** interface, as they do in the starter files.
- **`FacePamphletProfile`**—This class should encapsulate all the information for a single profile in the social network. Given a **`FacePamphletProfile`** object, you can find out that profile's name, associated image (or lack thereof), associated status (or lack thereof), and the list of names of friends for that profile.
- **`FacePamphletDatabase`**—This class keeps track of all the profiles in the FacePamphlet social network. Note that this class is completely separate from the user interface. It is responsible for managing profiles (adding, deleting, looking-up).
- **`FacePamphletCanvas`**—This class is a subclass of **`GCanvas`** that displays profiles as well as Application Messages on the display canvas. Note, however, that this class does **not** implement the **`ComponentListener`** interface. As a result, this canvas does **not** need to worry about updating the display as a result of window resizing. Of course, this is a feature you can add as a nice program extension if you like, but it's not required.

To help you with regard to developing your program in stages, we outline some development tasks below, along with more details regarding implementing the functionality provided in the program.

Task 1: Assemble the GUI interactors

As seen in the initial start-up screen of the application (shown below), there are a number of interactors (`JLabel`s, `JTextField`s, and `JButton`s) in both the `NORTH` and `WEST` border regions of the application.



Your first task is simply to add the interactors to the application window and create an implementation for the `actionPerformed` method that allows you to check whether you can detect button clicks and read what's in the text fields.

A few specific issues to note in the implementation of these interactors are the following:

- All text fields are `TEXT_FIELD_SIZE` characters wide. `TEXT_FIELD_SIZE` is just a constant set in `FacePamphletConstants`.
- The `Name` text field in the `NORTH` region does **not** have any `actionCommand` associated with it. In other words, pressing the Enter key in that text field should have no effect, so you don't need to worry about that case.
- The three text fields in the `WEST` region do have `actionCommand`s associated with them. The `actionCommand` associated with each respective text field should be the same as its corresponding button. For example, pressing the Enter key in the text field next to the `Change Status` button should have the same effect as pressing the `Change Status` button.
- If a text field is empty when its corresponding button is pressed, then **nothing** should happen. For example, if the `Name` text field in the `NORTH` region has nothing in it when the `Add` (or `Delete`, or `Lookup`) button is clicked (i.e., the text field's value is the empty string (`""`)), then we should simply not do anything as a result of the button click. This idea applies to all text fields in the application, and helps prevent situations such as trying to add a profile with an empty name, or trying to change the status of a profile to the empty string.

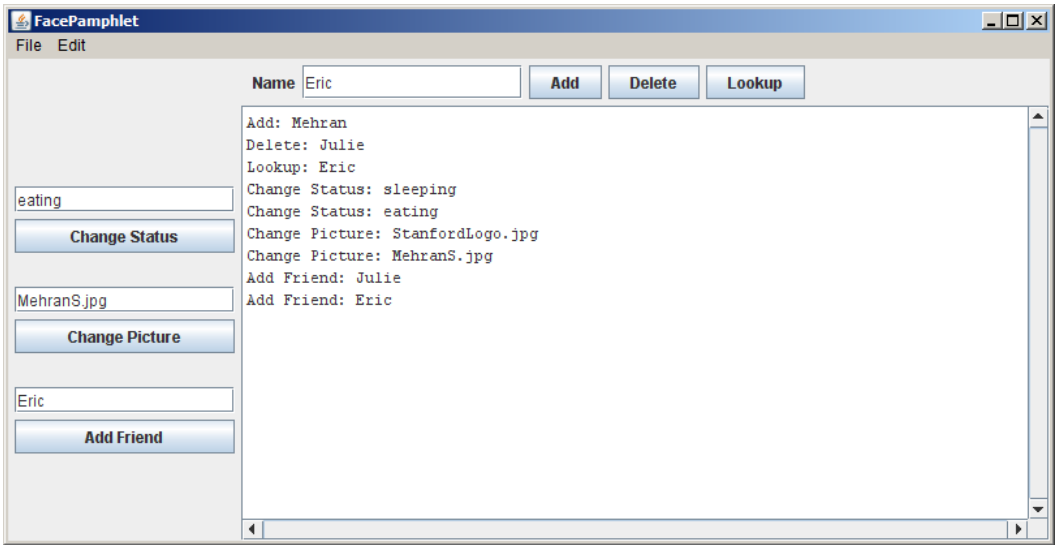
One issue to note is that in laying out the interactors in the **WEST** border region, you'll notice that there are spaces between the various text field/button pairs (for example, there is space between the **Change Status** button and the text field associated with **Change Picture**). These spaces should be produced by adding a **JLabel** with the label text **EMPTY_LABEL_TEXT** (this is just a constant defined in **FacePamphletConstants**) at the appropriate points when adding interactors to the **WEST** border region. So, your interactor layout code will likely include two lines at various points that look something like this:

```
add(new JLabel(EMPTY_LABEL_TEXT), WEST);
```

As you did on the previous assignment, you can take the strategy of changing the definition of the **FacePamphlet** class so that it extends **ConsoleProgram** instead of **Program**, at least for the moment. You can always change it back later. Once you have made that change, you can then use the console to record what's happening in terms of the interactors to make sure that you've got them right. For example, we provide below a transcript of the commands used to generate the output in Figure 1, in which the user has just completed the following actions:

1. Entered the name **Mehran** in the Name text field and clicked the **Add** button.
2. Entered the name **Julie** in the Name text field and clicked the **Delete** button.
3. Entered the name **Eric** in the Name text field and clicked the **Lookup** button.
4. Entered the text **sleeping** in the Change Status text field and clicked the **Change Status** button.
5. Entered the text **eating** in the Change Status text field and pressed the Enter key.
6. Entered the text **StanfordLogo.jpg** in the Change Picture text field and clicked the **Change Picture** button.
7. Entered the text **MehranS.jpg** in the Change Picture text field and pressed the Enter key.
8. Entered the text **Julie** in the Add Friend text field and clicked the **Add Friend** button.
9. Entered the text **Eric** in the Add Friend text field and pressed the Enter key.

Figure 1. Illustration of Task 1



Task 2: Implement the `FacePamphletProfile` class

The starter file for the **`FacePamphletProfile`** class appears in full as Figure 2 on the following pages. The starter file includes definitions for all of the public methods we expect you to define. The method definitions in the starter files, however, do nothing useful (they are just ***stubs***), although they occasionally include a **`return`** statement that gives back a default value of the required type. In Figure 2, for example, the **`getName`** method always returns the empty string ("") to satisfy the requirement that the method returns an **`String`** as defined in its header line.

The **`FacePamphletProfile`** class encapsulates the information pertaining to one profile in the social network. That information consists of four parts:

1. The name of the person with this profile, such as **`"Mehran Sahami"`** or **`"Julie Zelenski"`**
2. The status associated with this profile. This is just a **`String`** indicating what the person associated with the profile is currently doing. Until it is explicitly set, the status should initially be the empty string.
3. The image associated with that profile. This is a **`GImage`**. Until it is explicitly set, this field should initially be **`null`** since we don't initially have an image associated with a profile.
4. The list of friends of this profile. The list of friends is simply a list of the *names* (i.e., list of **`String`**s) that are friends with this profile. This list starts empty. The data structure you use to keep track of this list is left up to you.

The last method in the starter implementation of **`FacePamphletProfile`** is a **`toString`** method whose role is to return a human-readable representation of the data stored in the profile. The general form of the string returned by this method is:

name **`(`** ***status*** **`):`** ***comma separated list of friend names***

For example, if the variable **`profile`** contains the **`FacePamphletProfile`** data of a profile with name "Alice" whose status is "coding" and who has friends named Don, Chelsea, and Bob, then **`profile.toString()`** would return the string:

```
"Alice (coding): Don, Chelsea, Bob"
```

The **`toString`** method will be useful as you continue to develop your program in stages.

Figure 2. Starter file for the `FacePamphletProfile` class

```
/*
 * File: FacePamphletProfile.java
 * ------------------------------
 * This class keeps track of all the information for one profile
 * in the FacePamphlet social network.  Each profile contains a
 * name, an image (which may not always be set), a status (what
 * the person is currently doing, which may not always be set),
 * and a list of friends.
 */

import acm.graphics.*;
import java.util.*;


public class FacePamphletProfile implements FacePamphletConstants {

   /**
    * Constructor
    * This method takes care of any initialization needed for
    * the profile.
    */
   public FacePamphletProfile(String name) {
      // You fill this in
   }

   /** This method returns the name associated with the profile. */
   public String getName() {
      // You fill this in.  Currently always returns the empty string.
      return "";
   }

   /** This method returns the image associated with the profile.
    * If there is no image associated with the profile, the method
    * returns null. */
   public GImage getImage() {
      // You fill this in.  Currently always returns null.
      return null;
   }

   /** This method sets the image associated with the profile. */
   public void setImage(GImage image) {
      // You fill this in
   }

   /** This method returns the status associated with the profile.
    * If there is no status associated with the profile, the method
    * returns the empty string ("").
    */
   public String getStatus() {
      // You fill this in.  Currently always returns the empty string.
      return "";
   }

   /** This method sets the status associated with the profile. */
   public void setStatus(String status) {
      // You fill this in
```

```
    }

    /** This method adds the named friend to this profile's list of
     * friends.  It returns true if the friend's name was not already
     * in the list of friends for this profile (and the name is added
     * to the list).  The method returns false if the given friend name
     * was already in the list of friends for this profile (in which
     * case, the given friend name is not added to the list of friends
     * a second time.)
     */
    public boolean addFriend(String friend) {
        // You fill this in.  Currently always returns true.
        return true;
    }


    /** This method removes the named friend from this profile's list
     * of friends.  It returns true if the friend's name was in the
     * list of friends for this profile (and the name was removed from
     * the list).  The method returns false if the given friend name
     * was not in the list of friends for this profile (in which case,
     * the given friend name could not be removed.)
     */
    public boolean removeFriend(String friend) {
        // You fill this in.  Currently always returns false.
        return false;
    }


    /** This method returns an iterator over the list of friends
     * associated with the profile.
     */
    public Iterator<String> getFriends() {
        // You fill this in.  Currently always returns null.
        return null;
    }

    /** This method returns a string representation of the profile.
     * This string is of the form: "name (status): list of friends",
     * where name and status are set accordingly and the list of
     * friends is a comma separated list of the names of all of the
     * friends in this profile.
     *
     * For example, in a profile with name "Alice" whose status is
     * "coding" and who has friends Don, Chelsea, and Bob, this method
     * would return the string: "Alice (coding): Don, Chelsea, Bob"
     */
    public String toString() {
        // You fill this in.  Currently always returns the empty string.
        return "";
    }
}
```

Task 3: Implement the `FacePamphletDatabase` class

After you have defined the class **`FacePamphletProfile`**, you are ready to implement the **`FacePamphletDatabase`** class. The starter file for the **`FacePamphletDatabase`** class appears in full as Figure 3 on the following pages. As with the other files supplied with this assignment, the starter file includes definitions for all of the public methods we expect you to define.

The **`FacePamphletDatabase`** class is used to keep track of *all* the profiles in the social network. The class which contains five public entries:

- A constructor that has no parameters. You can use this to perform any initialization you may need for the database. Note: depending on how you implement the database, it is entirely possible that your constructor may not need to do anything. It's perfectly fine if that's the case.
- An **`addProfile`** method that is passed a **`FacePamphletProfile`**, and is responsible for adding that profile to the database. Note that **profile *names* are unique identifiers for profiles in the database**. In other words, no two profiles in the database should have the same name and the name associated with a profile will never change. So, when a new profile is being added, if there is already an existing profile with the same name, the existing profile should be replaced by the new profile. Note: depending on what data structure you use to keep track of the database, this behavior may actually be quite easy to implement.
- A **`getProfile`** method that takes a name, looks it up in the database of profiles, and returns the **`FacePamphletProfile`** with that name, or **`null`** if there is no profile with that name.
- A **`deleteProfile`** method that takes a profile name, and deletes the profile with that name from the profile database. Note that when we delete a profile from the database, we not only delete the profile itself, but we also update *all* other profiles in the database so as to remove the deleted profile's name from any friends lists in other profiles. In this way, we ensure that someone cannot be friends with a person who does not have a profile in the database.
- A **`containsProfile`** method that takes a profile name, and returns **`true`** if there is a profile with that name in the database. Otherwise, it returns **`false`**.

The code for this part of the assignment is not particularly difficult. The challenging part lies in figuring out how you want to represent the data so that you can implement the methods above as simply and as efficiently as possible.

Figure 3. Starter file for the `FacePamphletDatabase` class

```
/*
 * File: FacePamphletDatabase.java
 * -------------------------------
 * This class keeps track of the profiles of all users in the
 * FacePamphlet application.  Note that profile names are case
 * sensitive, so that "ALICE" and "alice" are NOT the same name.
 */

import java.util.*;

public class FacePamphletDatabase implements FacePamphletConstants {

   /** Constructor
    *  This method takes care of any initialization needed for
    *  the database.  */
   public FacePamphletDatabase() {
      // You fill this in
   }

   /** This method adds the given profile to the database.  If the
    *  name associated with the profile is the same as an existing
    *  name in the database, the existing profile is replaced by
    *  the new profile passed in.  */
   public void addProfile(FacePamphletProfile profile) {
      // You fill this in
   }

   /** This method returns the profile associated with the given name
    *  in the database.  If there is no profile in the database with
    *  the given name, the method returns null.  */
   public FacePamphletProfile getProfile(String name) {
      // You fill this in.  Currently always returns null.
      return null;
   }

   /** This method removes the profile associated with the given name
    *  from the database.  It also updates the list of friends of all
    *  other profiles in the database to make sure that this name is
    *  removed from the list of friends of any other profile.
    *
    *  If there is no profile in the database with the given name, then
    *  the database is unchanged after calling this method.  */
   public void deleteProfile(String name) {
      // You fill this in
   }

   /**
    * This method returns true if there is a profile in the database
    * that has the given name.  It returns false otherwise.  */
   public boolean containsProfile(String name) {
      // You fill this in.  Currently always returns false.
      return false;
   }
}
```

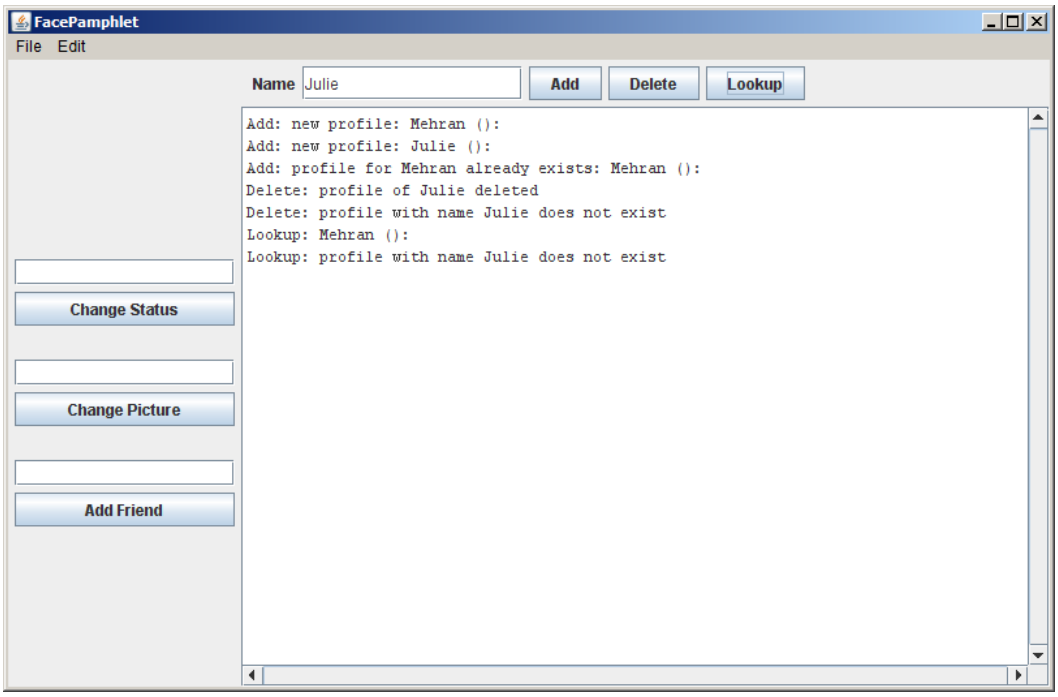
To test this part of the program, you can add code to the **FacePamphlet** program so that it creates the **FacePamphletDatabase** and then change the code for the **Add**, **Delete**, and **Lookup** button handlers as follows:

- Entering a name in the Name text field and clicking the **Add** button looks up the current name in the database to see if a profile with that name already exists. If the name does not exist, then it adds a new profile to the database and prints out "**Add: new profile: **" followed by the string version of the profile (using the **toString** method of the **FacePamphletProfile**). If the profile name already exists in the database, then it prints out the fact that the profile with that name already exists followed by the string representation of the profile.
- Entering a name in the Name text field and clicking the **Delete** button looks up the current name in the database to see if it exists. If the name does exist, then it deletes the profile with that name from the database and prints out that the profile was deleted. If the profile name does not exist in the database, then it simply prints out that a profile with the given name does not exist.
- Entering a name in the Name text field and clicking the **Lookup** button looks up the current name in the database to see if it exists. If the name does exist, then prints out "**Lookup: **" followed by the string version of the profile. If the name does not exist, then it prints out that a profile with the given name does not exist.

A sample run of this task is shown in Figure 4 (on the next page), where the user has just completed the sequence of actions given below. (Note that your text messages need not correspond exactly to those shown in the sample run here, as long as you can still verify that your program is working properly.)

1. Entered the name **Mehran** in the Name text field and clicked the **Add** button.
2. Entered the name **Julie** in the Name text field and clicked the **Add** button.
3. Again, entered the name **Mehran** in the Name text field and clicked the **Add** button.
4. Entered the name **Julie** in the Name text field and clicked the **Delete** button.
5. With the name **Julie** still in the Name text field, clicked the **Delete** button again.
6. Entered the name **Mehran** in the Name text field and clicked the **Lookup** button.
7. Entered the name **Julie** in the Name text field and clicked the **Lookup** button.

Figure 4. Illustration of Task 3



Task 4: Implement functionality for `Change Status`, `Change Picture`, and `Add Friend` buttons

The next step in the process is to complete more of the implementation of the **`FacePamphlet`** class, namely the functionality for the **`Change Status`**, **`Change Picture`**, and **`Add Friend`** buttons. The main issue to remember here is that these buttons effect the *current profile*, if there is one. As a result, one of the first things you should think about in implementing this task is how you will keep track of the current profile in the application. To help introduce the notion of the current profile, you might want to update the code for the **`Add`**, **`Delete`**, and **`Lookup`** button handlers so that:

- Whenever a new profile is added, the current profile is set to be the newly added profile. If the user tried to add a profile with the name of an existing profile, then the existing profile with that name is set to be the current profile (this is similar to the case below where the users simply looks up an existing profile).
- Whenever a profile is deleted (whether or not the profile to be deleted exists in the database), there is no longer a current profile (regardless of what the current profile previously was).
- Whenever the user lookups up a profile by name, the current profile is set to be the profile that the user looked up, if it exists in the database. If a profile with that name does not exist in the database, then there is no longer a current profile (regardless of what the current profile previously was).

Once you have a notion of a current profile implemented, then you are ready to actually implement the functionality for the **`Change Status`**, **`Change Picture`**, and **`Add Friend`** buttons.

Implementing `Change Status`

If the user enters some text in the text field associated with the **`Change Status`** button and either presses the **`Change Status`** button or hits Enter, the application should update as follows:

- If there is a current profile, then the status for that profile should be updated to the text entered, and you can just print out a message to that effect.

- If there is no current profile, then you should simply prompt the user to select a profile to change the status of (and there should be no changes to any of the profiles in the database).

Implementing `Change Picture`

If the user enters some text in the text field associated with the **`Change Picture`** button and either presses the **`Change Picture`** button or hits Enter, the application should update as follows:

- If there is a current profile, then we need to see if the we can create a **`GImage`** with the filename of the text entered in the text field. Checking to see if a valid image file exists can be accomplished using the code fragment below (where **`filename`** is a **`String`** containing the name of the image file we are trying to open):

  ```
  GImage image = null;
  try {
       image = new GImage(filename);
  } catch (ErrorException ex) {
       // Code that is executed if the filename cannot be opened.
  }
  ```

  Note in the code fragment above that the variable **`image`** will still have the value **`null`** if we were unable to open the image file with the given filename. Otherwise, the value of the variable **`image`** will be a valid **`GImage`** object (whose value will not be **`null`**).

  If we obtained a valid **`GImage`**, then the image for the current profile should be updated to this image, and you can print out a message to that effect (although you won't be able to display the actual image for now).

- If there is no current profile, then you should simply prompt the user to select a profile to change the image of (and there should be no changes to any of the profiles in the database).

In the starter bundle for this assignment we have provided you with an **`images`** folder that contains an initial set of images that you can use for this assignment. Of course, you can feel free to use your own image files as well (as long as they are in GIF or JPG format).

Implementing **`Add Friend`**

If the user enters some text in the text field associated with the **`Add Friend`** button and either presses the **`Add Friend`** button or hits Enter, the application should update as follows:

- If there is a current profile, then we need to see if the name entered in the text field is the name of a valid profile in the database. If it is, then we try to add the named friend to the list of friends for the current profile. If the named friend already exists in the list of friends for the current profile, then we simply write out a message that such a friend already exists. If that named friend does not previously exist in the list of friends (i.e., it was successfully added to the list of friends for the current profile), then (recalling that friendships are reciprocal) we also need to update the profile of the named friend to add the name of the current profile to its list of friends. For example, if the current profile was "Mehran" and we tried to add as a friend "Julie" (which, say, is the name of valid profile in the database, which is not already a friend of Mehran), then we should add Julie as a friend of Mehran and also add Mehran as a friend of Julie.

- If the name entered in the **`Add Friend`** text field is not a valid profile in the system, we should just print out a message to that effect.

- If there is no current profile, then you should simply prompt the user to select a profile to add a friend to (and there should be no changes to any of the profiles in the database).

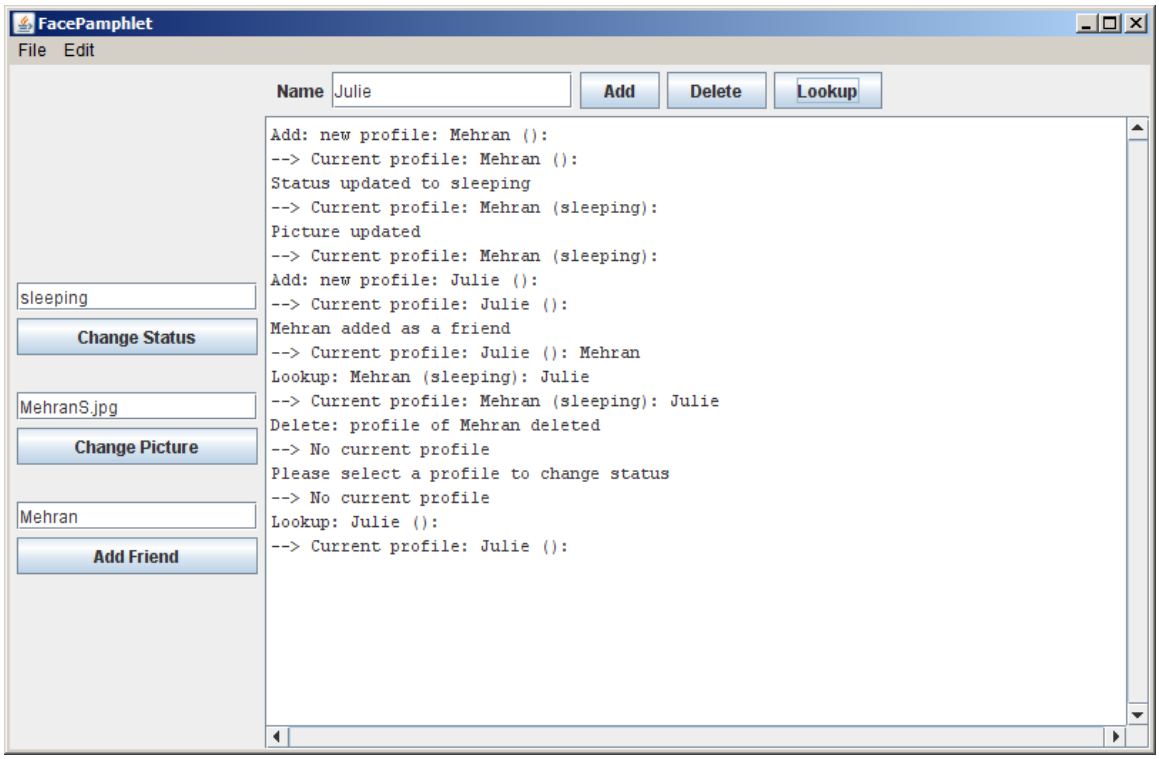
To show one possible example of the interactions at this task, we show a sample run in Figure 5 on the next page, where the user has just completed the sequence of actions given below. (Note that your text messages don't need to correspond exactly to those shown here, but you should be able to get the idea of what profile, if any, is the current profile at any given time, as well as the updates that are made to it.)

1. Entered the name **`Mehran`** in the Name text field and clicked the **`Add`** button. Note that at this point the current profile is set to Mehran's profile.
2. Entered the text **`sleeping`** in the Change Status text field and clicked the **`Change Status`** button.
3. Entered the text **`MehranS.jpg`** in the Change Picture text field and clicked the **`Change Picture`** button.
4. Entered the name **`Julie`** in the Name text field and clicked the **`Add`** button. Note that at this point the current profile is set to Julie's profile.

5. Entered the name **`Mehran`** in the Add Friend text field and clicked the **Add Friend** button. Note that Julie's current profile now shows Mehran as a friend.
6. Entered the name **`Mehran`** in the Name text field and clicked the **Lookup** button. Note that at this point the current profile is Mehran's profile and it now shows Julie as friend.

7. With the name **`Mehran`** still in the Name text field, we clicked the **Delete** button. Note that at this point there is no current profile.
8. With the text **`sleeping`** still in the Change Status text field, we clicked the **Change Status** button, and were prompted to select a profile since there is no current profile.
9. Entered the name **`Julie`** in the Name text field and clicked the **Lookup** button. Note that the current profile is now set to Julie's profile, and Mehran is no longer in her friend list since his profile was deleted previously.

Figure 5. Illustration of Task 4



Task 5: Implement the `FacePamphletCanvas` class and complete the implementation of the `FacePamphlet` class

At this point you actually have most of the functionality for keeping track of data in your social network application. All that's left is to create the actual graphical display of profiles, and then tie up a few loose ends to make sure you're displaying appropriate messages to the user.

The starter code for the **`FacePamphletCanvas`** class appears in Figure 6 on the next page. The class (which extends **`GCanvas`**) contains three public entries:

- A constructor that has no parameters. You can use this to perform any initialization you may need for the canvas. Note: depending on how you implement the canvas, it is entirely possible that your constructor may not need to do anything. It's perfectly fine if that's the case.
- A **`showMessage`** method that is passed a **`String`**, and is responsible for displaying that string as the Application Message at the bottom of the canvas. The method should display this Application Message text centered horizontally with respect to the width of the canvas, and the vertical baseline for the text should be located **`BOTTOM_MESSAGE_MARGIN`** pixels up from the bottom of the canvas. The font for the text should be set to **`MESSAGE_FONT`**. Note that **`BOTTOM_MESSAGE_MARGIN`** and **`MESSAGE_FONT`** are simply constants defined in **`FacePamphletConstants`**. Whenever this method is called, any previously displayed message is replaced with the new message text that is passed in.
- A **`displayProfile`** method that is passed a **`FacePamphletProfile`**, and is responsible for displaying the contents of that profile in the canvas, including the profile's name, image (if any), the status of the profile (if any), and the list of friends (if any). Whenever this method is called, all existing contents of the canvas should be cleared (including any previously displayed profile as well as any displayed Application Messages), and the profile passed in should be displayed. How the various components of the profile should be displayed is discussed in more detail below.

Figure 6. Starter file for the `FacePamphletCanvas` class

```
/*
 * File: FacePamphletCanvas.java
 * -----------------------------
 * This class represents the canvas on which the profiles in the social
 * network are displayed.  NOTE: This class does NOT need to update the
 * display when the window is resized.
 */

import acm.graphics.*;
import java.awt.*;
import java.util.*;

public class FacePamphletCanvas extends GCanvas
               implements FacePamphletConstants {

   /** Constructor
    *  This method takes care of any initialization needed for
    *  the display
    */
   public FacePamphletCanvas() {
      // You fill this in
   }

   /** This method displays a message string near the bottom of the
    *  canvas.  Every time this method is called, the previously
    *  displayed message (if any) is replaced by the new message text
    *  passed in.
    */
   public void showMessage(String msg) {
      // You fill this in
   }

   /** This method displays the given profile on the canvas.  The
    *  canvas is first cleared of all existing items (including
    *  messages displayed near the bottom of the screen) and then the
    *  given profile is displayed.  The profile display includes the
    *  name of the user from the profile, the corresponding image
    *  (or an indication that an image does not exist), the status of
    *  the user, and a list of the user's friends in the social
network.
    */
   public void displayProfile(FacePamphletProfile profile) {
      // You fill this in
   }

}
```

To start adding the graphical display code for profiles, you should go back to the **`FacePamphlet`** class and change its definition so that it extends **`Program`** rather than the temporary expedient of extending **`ConsoleProgram`** (as you may have been using during the tasks above). At the same time, you should remove the various **`println`** calls that allowed you to trace the operation of the interactors in the earlier tasks.

Now, you'll need to declare a **`FacePamphletCanvas`** private instance variable in your main **`FacePamphlet`** class:

```
private FacePamphletCanvas canvas;
```

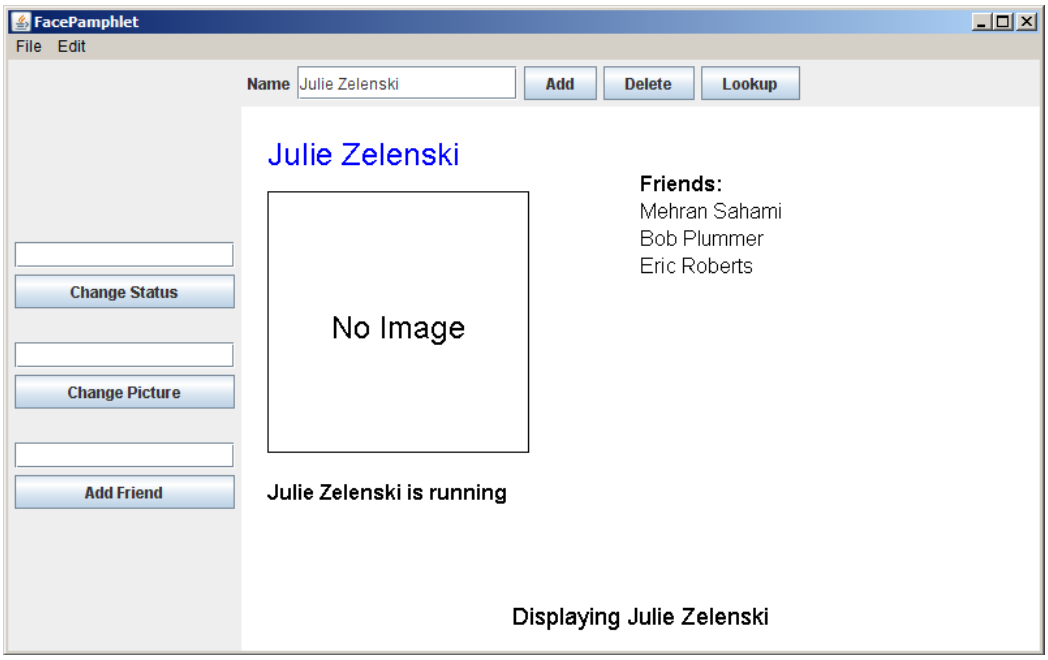
You should then change the constructor of the **`FacePamphlet`** class so that it creates a new **`FacePamphletCanvas`** object and adds that object to the display, as follows:

```
canvas = new FacePamphletCanvas();
add(canvas);
```

If you run the program with only these changes, it won't actually display anything on the canvas until you implement the methods of the **`FacePamphletCanvas`** class and call them from your **`FacePamphlet`** class.

Implementing `displayProfile`

Much of the layout for graphical display of profiles is dictated by constant values defined in **`FacePamphletConstants`**. Here we explain how each component of the profile display should be set up, and refer to the sample screen below as necessary:



- **Name**: Near the top of the display, the name associated with the profile ("Julie Zelenski" in the example above) should be displayed in the color Blue. Horizontally, the text should located **`LEFT_MARGIN`** pixels in from the left-hand side of the canvas. Vertically, the *top* of the text (not its baseline) should be **`TOP_MARGIN`** pixels from the top of the canvas. The font for the text should be set to **`PROFILE_NAME_FONT`**.

- **Image**: Although there is currently no image associated with the profile above, we can see that there is space set aside to display an image immediately under the name of the profile. The space for the image will always be **`IMAGE_WIDTH`** by **`IMAGE_HEIGHT`** pixels.

  If no image is associated with the profile then a rectangle of the dimensions of the image size should be drawn. Horizontally, this rectangle should be located **`LEFT_MARGIN`** pixels in from the left-hand side of the canvas. Vertically, the top of the rectangle should be should be **`IMAGE_MARGIN`** pixels below the baseline of the profile name text. Centered (both horizontally and vertically) within this rectangle should be the text "No Image" in the font **`PROFILE_IMAGE_FONT`**.

  If an image is associated with the profile then the image should be displayed (in the same location as the rectangle described above). The image should be scaled so that it displays with **`IMAGE_WIDTH`** by **`IMAGE_HEIGHT`** pixels. The **`scale`** method of **`GImage`** should be useful to make image display with the appropriate size.

- **Status**: Under the area for the image, the current status of the person with this profile should be displayed (Julie's status is "running" in the example above). If the profile currently has no status (i.e., it has an empty status string), the text "No current status" should be displayed. If the profile does have a status, the status text should have the name of the profile followed by the word "is" and then the status text for the profile. In any case, the line describing the profile's status should be located horizontally **`LEFT_MARGIN`** pixels in from the left-hand side of the canvas. Vertically, the *top* of the text (not its baseline) should be located **`STATUS_MARGIN`** pixels below the bottom of the image. The font for the text should be set to **`PROFILE_STATUS_FONT`**.

- **Friends**: To the right of the profile's name, there is the header text "Friends:", and the names of the friends of this profile (e.g., Mehran Sahami, Bob Plummer, and Eric Roberts) are listed below. The start of the header text "Friends:" should be horizontally located at the midpoint of width of the canvas. Vertically, the *baseline* for this text should be the same as the top of the image area. The "Friends:" header text should be displayed in the font **`PROFILE_FRIEND_LABEL_FONT`**. Immediately below the header, the friends of this profile should be listed sequentially, one per line, with the same horizontal location as the "Friends:" header text. You can use the **`getHeight()`** method of **`GLabel`** to determine how to vertically space out the list of friends to get one friend per line. The friend names should be displayed in the font **`PROFILE_FRIEND_FONT`**.

Note that you don't need to worry about long friend lists that may overwrite a long status message that the profile may have. This might make for an interesting extension, but is certainly not something you need to worry about for this assignment.

- **Application Message**: As described previously (but repeated here for completeness) the Application Message text ("Displaying Julie Zelenski" in the example above) should be centered with respect to the width of the canvas, and the baseline for the text should be located `BOTTOM_MESSAGE_MARGIN` pixels up from the bottom of the canvas. The font for the text should be set to `MESSAGE_FONT`.

To initally work on implementing `displayProfile` it might be easiest to simply put a single call to this method (of the `canvas`) in the code that where you add a new profile to the social network. In this way, when you start your application, you can simply try adding the first profile and see if things display correctly (at least for the initial empty profile). Once you get that working then you can wire up the rest of your program.

Finishing Up

In finishing up the program, you need to make calls at appropriate times to `displayProfile` and `showMessage` in your `FacePamphlet` class. Below we outline the behavior you should produce in your application. If you have any questions, you can always refere to the demo applet on the class web site to see how various situations should be handled.

- **Adding a Profile**

  When a new profile is being added you should see if a profile with that name already exists. If it does, you should display the existing profile and give the user the message "A profile with the name ***<name>*** already exists". If the profile does not already exist, you should display the newly created profile and give the user the message "New profile created".

- **Deleting a Profile**

  When a profile is being deleted you should see if a profile with that name exists. If it does, you should delete the profile, clear any existing profile from the display, and give the user the message "Profile of ***<name>*** deleted". If the profile does not exist, you should clear any existing profile from the display, and give the user the message "A profile with the name ***<name>*** does not exist".

- **Looking up a Profile**

  When a profile is being looked up you should see if a profile with that name exists. If it does, you should display the profile, and give the user the message "Displaying ***<name>***". If the profile does not exist, you should clear any existing profile from the display, and give the user the message "A profile with the name ***<name>*** does not exist".

- **Changing Status**

  When the status for a profile is being changed, you should determine if there is a current profile. If no current profile exists, you should just give the user the message "Please select a profile to change status". If there is a current profile, you should update its status, redisplay the profile (to show the changed status), and give the user the message "Status updated to ***<status>***".

- **Changing Picture**

  When the picture for a profile is being changed, you should determine if there is a current profile. If no current profile exists, you should just give the user the message "Please select a profile to change picture". If there is a current profile, you should see if the filename given for the picture contains a valid image, and if it does, you should add the image to the profile, redisplay the current profile (to show the new image), and give the user the message "Picture updated". If the given filename could not be opened, you should just give the user the message "Unable to open image file: ***<filename>***". In that case, the image associated with the profile is unchanged.

- **Adding Friend**

  When a friend is being added to a profile, you should determine if there is a current profile. If no current profile exists, you should just give the user the message "Please select a profile to add friend". If there is a current profile, you should see if the given friend name is the name for a valid profile in the social network. If the name is valid and the current profile does not already have that person as a friend, then you should update the friend list for both the current profile and the named friend, redisplay the current profile (to show the addition of the friend), and give the user the message "***<friend name>*** added as a friend". If the named friend is already a friend of the current profile, you should just display the message "***<name of current profile>*** already has ***<friend name***> as a friend." If the named friend does not have a profile in the social network, then you should simply display the message "***<friend name>*** does not exist."

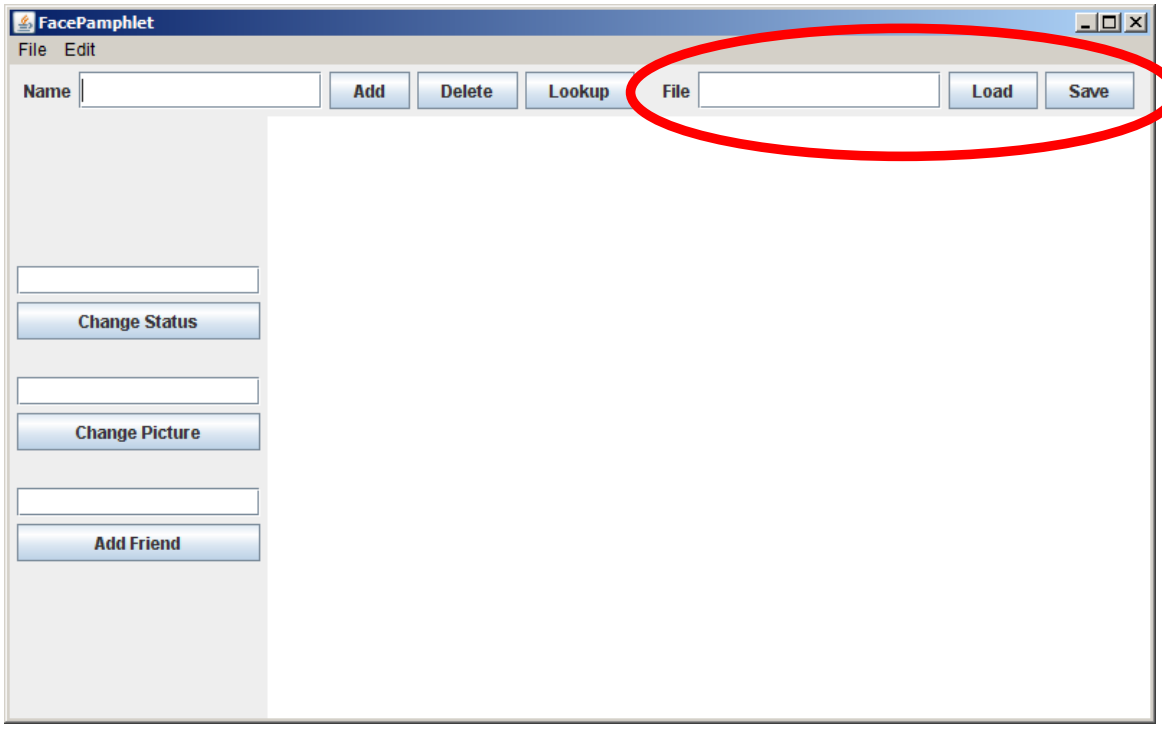
Congratulations! Once you've gotten this working, you've just finished implementing your very own social network application.

Task 6: Implement Loading and Saving social networks from a file

One of the most useful extensions you could add to your social network is the ability to load and save the contents of the social network to a file, so that you don't need to recreate the whole network from scratch every time you run your application. Since this is a particularly useful extension, we outline some steps below to guide you in adding such an extension.

Adding interactors to support Load/Save functionality

The first step in extending your program to support Load/Save functions it to augment the user interface with additional interactors that allow the user to specify the name of the file for the social network to be Loaded from or Saved to. This is most easily accomplished by adding a new text field (along with an appropriate label) in the `NORTH` region followed by two buttons for **Load** and **Save** respectively, as indicated in the interface below:



Note that you should **not** allow a user to Load/Save a file with the empty string (`""`) as a file name. In other words, if the **File** text field is empty, you should simply ignore clicks on the **Load** and **Save** buttons (in the same way that users cannot create a profile that has the empty string as a name).

Loading a network

When the user enters a file name in the text field labeled **File** and presses the **Load** button, you should attempt to load (i.e., read) a file with the given name that contains all the data for an entire social network. Note that you will likely need to add a `public` method to the `FacePamphletDatabase` class that is responsible for reading a data file and storing its contents. And remember to add:

```
import java.io.*;
```

to any class where you are doing file operations. Your program should first attempt to see if the file specified by the user exists. If it does not, the program should not do anything to the existing social network, and should simply report (as an Application Message) that it was "Unable to open file ***<filename>***". If the file does exist, you should clear all the contents (profiles) from the current social network and then load a new network based on the contents of the file. You can assume that the input file is properly formatted (i.e., you don't need to do any error checking on the contents of the file, unless you really want to). After you load the data for the new social network, you should clear the current profile display, and simply display the Application Message "Loaded file ***<filename>***".

Social network file format

The data file containing the specification of the social network starts with a line that contains the total number of profiles in the network. This is followed by the contents of each profile in the network, formatted as follows:

`Profile name`
`Name of image file for profile` (this will be a blank line if there is no image)
`Status of the profile` (this will be a blank line if there is no current status)
`The names of the friends of this profile` (listed one name per line, if any)
A blank line denoting the end of this profile (to separate it from the next profile)

A sample input file named `sample-network.txt` is shown at the top of the next page. This file represents a simple social network containing four profiles, named "Wilma Flintstone", "Fred Flintstone", "Barney Rubble" and "Dino Dogasaurus". Dino Dogasaurus is friends with both Wilma Flintstone and Fred Flintstone, and there are no other friendships in the network.

Note that in the example file below, we also list comments in ***bold italic*** font that explain each line in the file. These comments would <u>not</u> actually appear in the data file.

You can assume that the names of any image files in the data file that you are loading are refering to valid image files that you can display.

File: `sample-network.txt`

	Explanation of lines in data file
`4`	← *There are 4 profiles in this network*
`Wilma Flintstone`	← *Name of the first profile is "Wilma Flintstone"*
`WilmaF.jpg`	← *Wilma Flintstone's profile has image file "WilmaF.jpg"*
	← *Wilma Flintstone's profile does not have its Status set*
`Dino Dogasaurus`	← *Wilma Flintstone is friends with Dino Dogasaurus*
	← *Blank line denoting end of this profile*
`Fred Flintstone`	← *Name of the next profile is "Fred Flintstone"*
`FredF.jpg`	← *Fred Flintstone's profile has image file "FredF.jpg"*
`coding like a fiend`	← *Fred Flintstone's status is "coding like a fiend"*
`Dino Dogasaurus`	← *Fred Flintstone is friends with Dino Dogasaurus*
	← *Blank line denoting end of this profile*
`Barney Rubble`	← *Name of the next profile is "Barney Rubble"*
	← *Barney Rubble's profile does not have an image*
	← *Barney Rubble's profile does not have its Status set*
	← *Blank line at end of profile (Note: Nick has no friends)*
`Dino Dogasaurus`	← *Name of the next profile is "Dino Dogasaurus"*
	← *Dino Dogasaurus' profile does not have an image*
`working on another book`	← *Dino Dogasaurus' status is "working on another book"*
`Fred Flintstone`	← *Dino Dogasaurus is friends with Fred Flintstone*
`Wilma Flintstone`	← *Dino Dogasaurus is friends with Wilma Flintstone*
	← *Blank line denoting end of this profile*

Saving a network

When the user enters a file name in the text field labeled **File** and presses the **Save** button, you should save (i.e., write) to a file the data for the entire current social network. Similarly to the case of Loading, you will likely need to add a `public` method to the `FacePamphletDatabase` class that is responsible for saving a data file. You should write out a data file that matches the file format described above. After you save the data for the current social network, you should display the Application Message "Saved file ***<filename>***". You need not clear the current profile being displayed in this case.

Note that other than the issue of actually writing out the data file, implementing the Save functionality has some (small) implications for other changes that are needed in your program. Specifically, in the basic version of FacePamphlet, when a user entered the name of an image file to display, you likely only stored the actual `GImage` corresponding to that image in the `FacePamphletProfile` object (and not the corresponding file name entered by the user). In order to support Saving profiles, you will now need to modify your program to also store the name of the image file (in addition to the actual `GImage`) in a `FacePamphletProfile`. This may require adding additional "getter" and "setter" methods to the `FacePamphletProfile` class. Storing the name of the image file will enable you to write it out to the data file when the user saves the social network.

As a side note, when writing a data file, you can throw a new `ErrorException` in the case where any writing operations fail (say, when you catch an `IOException`). To use `ErrorException`, remember to import the following package in your code:

```
import acm.util.*;
```

If you would like to add even more extensions to your program after getting the Load/Save extended functionality working, there are several suggestions for additional extensions below. And, of course, if you do add further extensions, feel free to change the format of the data file for Loading/Saving social networks as needed in order to allow you to save additional information relevant to your extensions.

Additional extension ideas

Here are some additional ideas for ways to extend your FacePamphlet program:

- *Keep track of additional information for each profile.* The current profile only keeps track of a name, image, status and a list of friends. In real social networks, there is much more information about users that is kept track of in profiles (e.g., age, gender, where they may have gone to school, etc.) Use your imagination. The more challenging issue will be how you appropriately display this additional information graphically in the profile display.
- *Make the names of friends clickable "Lookup" links.* In the list of friends for a profile, you could implement the ability to click on the name of a friend (i.e., detect a click on the `GLabel`) and then "Lookup" the profile for the friend's name that was clicked on. This helps to make navigation through the profiles in the social network easier.
- *Support for communities.* Many social networking applications allow for keeping track of "communities" (or groups) that profiles can belong to. In many ways, being a member of a community is similar to having that community as a "friend"—a community has a list of members (similar to a list of friends for a profile) and each profile can be a member of many communities (much in the same way that a profile can have many friends). Adding support for communities would help make your social network more realistic and may not actually require too much work if you can leverage some of the conceptual similarities with respect to communities being like "friends".
- *Finding friends of friends.* Another interesting aspect of social networks is not only keeping track of how many people you have as friends, but also how quickly that number grows as you consider all the friends of your friends, and their friends, and so on. Displaying these sorts of properties of the social network are neat features that show just how few degrees of separtion there are between people. Along these same lines, it would be interesting to find and display "friendship chains" that show the shortest sequence of friendship relations that create a chain from one profile to another. For example, if X is a friend of Y, and Y is a friend of Z, then a friendship chain exist that goes: X → Y → Z. Finding longer chains can be a fun and challenging problem.
- *Adjust the profile display as the application window is resized.* You got some practice with this already with the NameSurfer application and it would be an interesting extension to apply some of those same ideas here. The more challenging issue is how you would decide to change font sizes and the size of the image as the display size grew or shrank.
- *Have Fun!* There's really no shortage of ways that you could extend your FacePamphlet application. In fact, whole companies have been started based on creating a social network application with some cool new features.