Mehran Sahami
CS 106A

Handout #39
November 14, 2007

Assignment #6—NameSurfer
Due: 3:15pm on Wednesday, November 28th

The NameSurfer assignment was created by Nick Parlante and further revised by Patrick Young and Eric Roberts

This assignment has two primary goals. The first is to give you an opportunity to use Java interactors to create an application that looks more like a modern interactive program complete with buttons, text fields, and a resizable graphical display. The second goal is to pull together the various facilities you have learned about so far to create an interesting application that—unlike Breakout, Hangman, and Yahtzee—is not a game but rather a useful program that presents data on some fascinating sociological questions.

Overview of the NameSurfer project

Against all bureaucratic stereotypes, the Social Security Administration, provides a neat web site showing the distribution of names chosen for children over the last 100 years in the United States (**http://www.ssa.gov/OACT/babynames/**). The Social Security Administration provides data that shows the 1000 most popular boy and girl names for children at 10 year intervals. The data can be boiled down to a single text file that looks something like this:

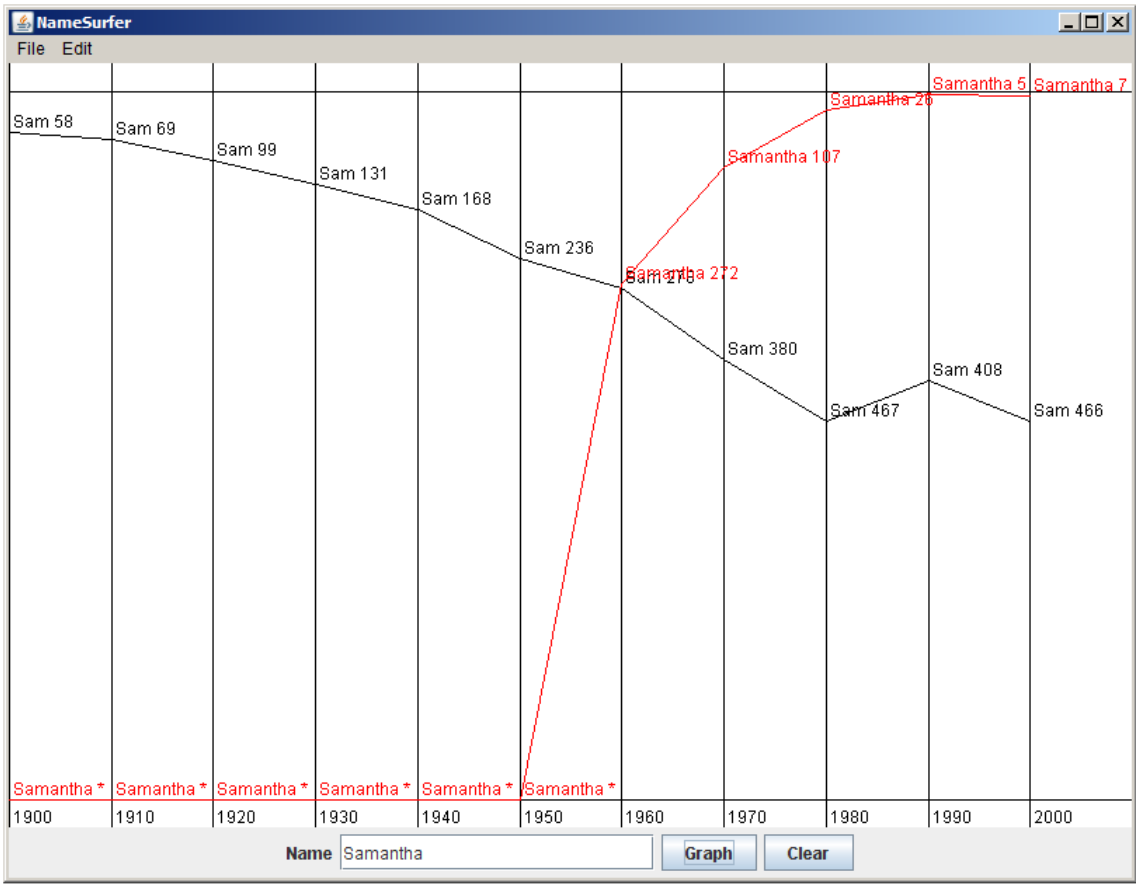
NamesData.txt

```
. . .
Sam 58 69 99 131 168 236 278 380 467 408 466
Samantha 0 0 0 0 0 0 272 107 26 5 7
Samara 0 0 0 0 0 0 0 0 0 0 886
Samir 0 0 0 0 0 0 0 0 920 0 798
Sammie 537 545 351 325 333 396 565 772 930 0 0
Sammy 0 887 544 299 202 262 321 395 575 639 755
Samson 0 0 0 0 0 0 0 0 0 0 915
Samuel 31 41 46 60 61 71 83 61 52 35 28
Sandi 0 0 0 0 704 864 621 695 0 0 0
Sandra 0 942 606 50 6 12 11 39 94 168 257
. . .
```

Each line of the file begins with the name, followed by the rank of that name in each of the 11 decades since 1900, counting the current one: 1900, 1910, 1920, and so on up to 2000. A rank of 1 indicates the most popular name that year, while a rank of 997 indicates a name that is not very popular. A 0 entry means the name did not appear in the top 1000 names for that year and therefore indicates a name that is even less popular. The elements on each line are separated from each other by a single space. The lines happen to be in alphabetical order, but nothing in the assignment depends on that fact.

As you can see from the small excerpt from the file, the name Sam was #58 in the first decade of the 1900s and is slowly moving down. Samantha popped on the scene in the 1960s (possibly because the show *Bewitched*, which had a main character names Samantha ran on television during those years) and is moving up strong to #7. Samir barely appears in the 1980s (at rank #920), but by the current decade is up to #798. The database counts children born in the United States, so trends in particular names tend to reflect the evolution of ethnic communities over the years.

Figure 1. Sample run of the NameSurfer program (with names "Sam" and "Samantha")



The goal of this assignment is to create a program that graphs these names over time, as shown in the sample run in Figure 1. In this diagram, the user has just typed **`Samantha`** into the box marked "Name" and then clicked on the "Graph" button, having earlier done exactly the same thing for the name **`Sam`**. Whenever the user enters a name, the **`NameSurfer`** program creates a new plot line showing how that name has fared over the decades. Clicking on the "Clear" button removes all the plot lines from the graph so that the user can enter more names without all the old names cluttering up the display.

To give you more experience working with classes that interact with one another, the **`NameSurfer`** application as a whole is broken down into several separate class files, as follows:

- **`NameSurfer`**—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 buttons at the bottom of the window, but only to the point of redirecting those events to the objects represented by the other classes.
- **`NameSurferConstants`**—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 **`NameSurferConstants`** interface, as they do in the starter files. The **`NameSurferConstants`** interface therefore has the same role that **`YahtzeeConstants`** did in Assignment #5.

- **`NameSurferEntry`**—This class ties together all the information for a particular name. Given a **`NameSurferEntry`** object, you can find out what name it corresponds to and what its popularity rank was in each decade.
- **`NameSurferDataBase`**—This class keeps track of all the information stored in the data files, but is completely separate from the user interface. It is responsible for reading in the data and for locating the data associated with a particular name.
- **`NameSurferGraph`**—This class is a subclass of **`GCanvas`** that displays the graph of the various names by arranging the appropriate **`GLine`** and **`GLabel`** objects on the screen, just as with the various graphical programs you've written this quarter.

Even though the class structure sounds complicated, the **`NameSurfer`** application code is about the same size as **`Yahtzee`**. Even if the scale of the project is comparable to the last assignment, the wise course is to start on the assignment soon and keep up with the milestones described in this handout.

Milestone 1: Assemble the GUI interactors

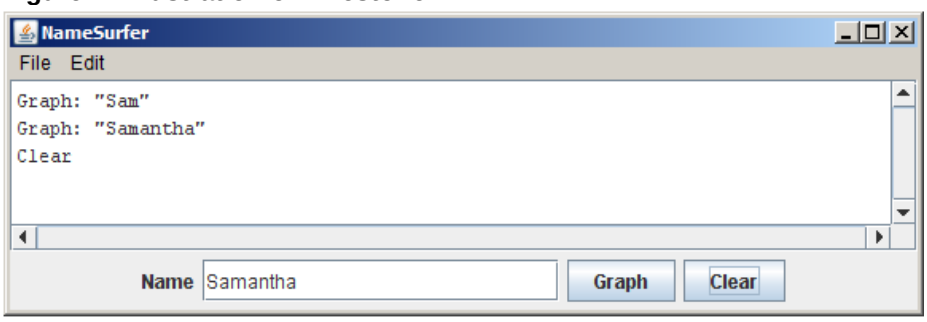
If you look at the bottom of Figure 1, you will see that the region along the **`SOUTH`** edge of the window contains several interactors: a **`JLabel`**, a **`JTextField`**, and three **`JButton`**s. Since putting up interactors is something you haven't done in previous assginments, you probably want to work on this step before it becomes complicated with all the other parts of the assignment. Thus, your first milestone is simply to add the interactors to the 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 field.

The simplest strategy to check whether your program is working is to change the definition of the **`NameSurfer`** 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, Figure 2 shows a possible transcript of the commands used to generate the output from Figure 1, in which the user has just completed the following actions:

1. Entered the name **`Sam`** in the text field and clicked the **Graph** button.
2. Entered the name **`Samantha`** in the text field and then typed the ENTER key.
3. Clicked the **Clear** button.

The hard part about reaching this milestone is understanding how interactors work. Once you do, writing the code is quite straightforward – it's only 10 to 15 lines of code.

Figure 2. Illustration of Milestone 1



Milestone 2: Implement the `NameSurferEntry` class

The starter file for the **`NameSurferEntry`** class appears in full as Figure 3 on the following page. 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 method definitions in the starter files, however, do nothing useful, although they occasionally include a **`return`** statement that gives back a default value of the required type. In Figure 3, for example, the **`getRank`** method always returns 0 to satisfy the requirement that the method returns an **`int`** as defined in its header line.

Methods that will eventually become part of the program structure but that are temporarily unimplemented are called **stubs**. Stubs play a very important role in program development because they allow you to set out the structure of a program even before you write most of the code. As you implement the program, you can go through the code and replace stubs with real code as you need it.

The **`NameSurferEntry`** class encapsulates the information pertaining to one name in the database. That information consists of two parts:

1. The name itself, such as **`"Sam"`** or **`"Samantha"`**
2. A list of 11 values indicating the rank of that name in each of the decades from 1900 to 2000, inclusive

The class definition begins with a constructor that creates an entry from the line of data that appears in the **`NamesData.txt`** file. For example, the entry for **`Sam`** looks like this:

```
Sam 58 69 99 131 168 236 278 380 467 408 466
```

The idea behind the design of this constructor is that it should be possible to read a line of data from the file and then create a new entry for it using code that looks like this:

```
String line = rd.readLine();
NameSurferEntry entry = new NameSurferEntry(line);
```

The implementation of the constructor has to divide up the line at the spaces, convert the digit strings to integers (using **`Integer.parseInt`**), and then store all of this information as the private state of the object in such a way that it is easy for the **`getName`** and **`getRank`** methods to return the appropriate values.

The last method in the starter implementation of **`NameSurferEntry`** is a **`toString`** method whose role is to return a human-readable representation of the data stored in the entry. For example, if the variable **`entry`** contains the **`NameSurferEntry`** data for **`Sam`**, you might want **`entry.toString()`** to return a string like this:

```
"Sam [58 69 99 131 168 236 278 380 467 408 466]"
```

Defining **`toString`** for a class has the wonderful advantage that it makes it possible to print out objects of that class using **`println`**, just as you do for primitive values. Whenever Java needs to convert an object to a string, it always calls its **`toString`** method to do the job. The default definition of **`toString`** in the **`Object`** class doesn't supply much useful information, and you will find that your debugging sessions get much easier if you can look easily at the values of your objects.

To show that you've got **NameSurferEntry** implemented correctly, you might want to write a very simple test program that creates an entry from a specific string and then verifies that the other methods work as they are supposed to.

Figure 3. Starter file for the NameSurferEntry class

```
/*
 * File: NameSurferEntry.java
 * --------------------------
 * This class represents a single entry in the database.  Each
 * NameSurferEntry contains a name and a list giving the popularity
 * of that name for each decade stretching back to 1900.
 */

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

public class NameSurferEntry implements NameSurferConstants {

/**
 * Creates a new NameSurferEntry from a data line as it appears
 * in the data file.  Each line begins with the name, which is
 * followed by integers giving the rank of that name for each
 * decade.
 */
   public NameSurferEntry(String line) {
      // You fill this in //
   }

/**
 * Returns the name associated with this entry.
 */
   public String getName() {
      // You need to turn this stub into a real implementation //
      return null;
   }

/**
 * Returns the rank associated with an entry for a particular
 * decade.  The decade value is an integer indicating how many
 * decades have passed since the first year in the database,
 * which is given by the constant START_DECADE.  If a name does
 * not appear in a decade, the rank value is 0.
 */
   public int getRank(int decade) {
      // You need to turn this stub into a real implementation //
      return 0;
   }

/**
 * Returns a string that makes it easy to see the value of a
 * NameSurferEntry.
 */
   public String toString() {
      // You need to turn this stub into a real implementation //
      return "";
   }
}
```

Milestone 3: Implement the `NameSurferDataBase` class

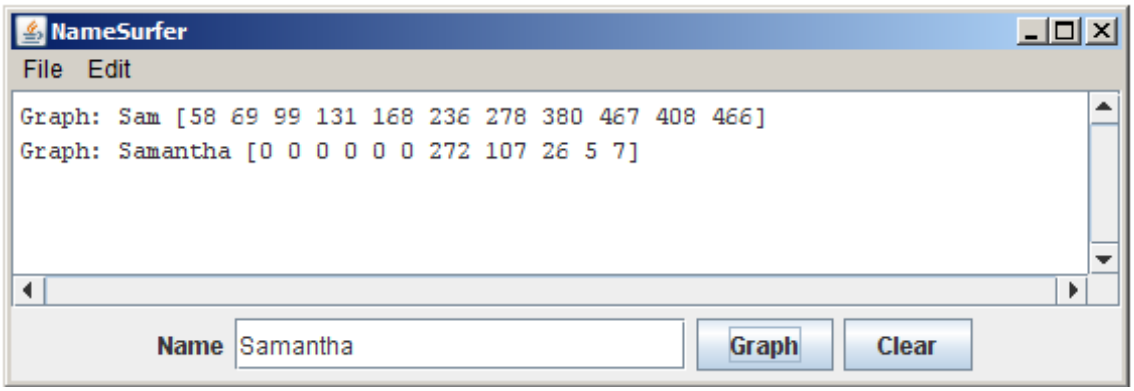
The next step in the process is to implement the **`NameSurferDataBase`** class, which contains two public entries:

- A constructor that takes the name of a data file and uses that to read in the entire set of data from the file into internal data structures that allow the class to keep track of all the records as a database.
- A **`findEntry`** method that takes a name, looks it up in the stored database, and returns the **`NameSurferEntry`** for that name, or **`null`** if that name does not appear.

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 **`findEntry`** method as simply and as efficiently as possible.

To test this part of the program, you can add a line of code or two to the **`NameSurfer`** program so that it creates the **`NameSurferDataBase`** and then change the code for the button handlers so that clicking the "Graph" button looks up the current name in the data base and then displays the corresponding entry (using its **`toString`** method), as shown in Figure 4 below.

Figure 4. Illustration of Milestone 3



Milestone 4: Create the background grid for the `NameSurferGraph` class

The next step in the process is to begin the implementation of the **`NameSurferGraph`** class, which is responsible for displaying the graph in the window by building the underlying model. The starter code for the **`NameSurferGraph`** class appears in Figure 5 on the next page.

Figure 5. Starter file for the `NameSurferGraph` class

```
/*
 * File: NameSurferGraph.java
 * --------------------------
 * This class is responsible for updating the graph whenever the
 * list of entries changes or the window is resized.
 */

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

public class NameSurferGraph extends GCanvas
    implements NameSurferConstants, ComponentListener {

/**
 * Creates a new NameSurferGraph object that displays the data.
 */
      public NameSurferGraph() {
            addComponentListener(this);
            // You fill in the rest //
      }

/**
 * Clears the list of name surfer entries stored inside this class.
 */
      public void clear() {
            // You fill this in //
      }

/**
 * Adds a new NameSurferEntry to the list of entries on the display.
 * Note that this method does not actually draw the graph, but
 * simply stores the entry; the graph is drawn by calling update.
 */
      public void addEntry(NameSurferEntry entry) {
            // You fill this in //
      }

/**
 * Updates the display image by deleting all the graphical objects
 * from the canvas and then reassembling the display according to
 * the list of entries.  Your application must call update after
 * calling either clear or addEntry; update is also called whenever
 * the size of the canvas changes.
 */
      public void update() {
            // You fill this in //
      }

/* Implementation of the ComponentListener interface */
      public void componentHidden(ComponentEvent e) { }
      public void componentMoved(ComponentEvent e) { }
      public void componentResized(ComponentEvent e) { update(); }
      public void componentShown(ComponentEvent e) { }

}
```

There are a couple of important items in the **NameSurferGraph** starter file that are worth noting:

1. This class extends **GCanvas**, which means that every **NameSurferGraph** object is not only a **GCanvas** but also an instance of all the superclasses of the **GCanvas** class. **GCanvas** is a subclass of **Component** in the standard **java.awt** package and therefore is part of the hierarchy of classes that can be added to the display area of a program. Moreover, it means we can call any of the **GCanvas** methods, such as adding or removing **GObject**s, from within **NameSurferGraph**.
2. The starter file includes a tiny bit of code that monitors the size of the window and calls **update** whenever the size changes. This code requires only a couple of lines to implement, but would be hard to explain well enough for you to implement on your own. Writing a page of description so that you could add a couple of lines seemed like overkill, particularly given that the strategy is easiest to learn by example.

To start the process of adding the graphing code, go back to the **NameSurfer** class and change its definition so that it extends **Program** rather than the temporary expedient of extending **ConsoleProgram** (if you were using that for debugging). At the same time, you should remove the various **println** calls that allowed you to trace the operation of the interactors in the earlier milestones.

Now, you'll need to declare a **NameSurferGraph** private instance variable in your main **NameSurfer** class:

```
private NameSurferGraph graph;
```

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

```
graph = new NameSurferGraph();
add(graph);
```

If you run the program with only these changes, it won't actually display anything on the screen. To create the graph, you need to implement the **update** method, which will almost certainly involve defining private helper methods as well. As a first step, write the code to create the background grid for the graph, which consists of the vertical line separating each decade, the horizontal lines that provide space for the top and bottom borders (which are there to ensure that the text labels stay within the window bounds), and the labels for the decades. As with all the graphical applications you've written, the lines and labels are represented using **GLine** and **GLabel** objects, which you add to the graph in the appropriate positions.

Milestone 5: Complete the implementation of `NameSurferGraph` class

In addition to creating the background grid, the **`update`** method in **`NameSurferGraph`** also has to plot the actual data values. As you can see from Figure 5, the **`NameSurferGraph`** class includes two methods for specifying what entries are displayed on the screen. The **`addEntry`** method adds a new **`NameSurferEntry`** to a list of values that are currently on the display. The **`clear`** method deletes all of the entries in that list so as to clear the graph.

It is important to note that neither **`addEntry`** or **`clear`** actually changes the display. To make changes in the display, you need to call **`update`**, which deletes any existing **`GObject`**s from the canvas and then assembles everything back up again. At first glance, this strategy might seem unnecessary. It would, of course, be possible to have **`addEntry`** just add all of the **`GLine`**s and **`GLabel`**s necessary to draw the graph.

The problem with that approach is that it would no longer be possible to reconstruct the entire graph. In this example, you need to do just that to create a new graph whenever you change the size of the display. By storing all of the entries in an internal list, the **`NameSurferGraph`** class can redraw everything when **`update`** is invoked from the **`componentResized`** method.

There are a couple of points that you should keep in mind while implementing this part of the program:

- To make the data easier to read, the lines on the graph are shown in different colors. In the sample applet on the web site, the first data entry is plotted in black, the second in red, the third in blue, and the fourth in magenta. After that, the colors cycle around again through the same sequence.
- The fact that rank 1 is at the top of the window and rank 1000 is at the bottom means that it can sometimes seem confusing that rank 0—which means that the name does not appear in the top 1000 values for that year—appears at the bottom. To reduce the apparent discontinuity between rank 1 and rank 0, the entries for names that are absent from the data for a decade are **listed with an asterisk** instead of a numeric rank. You can see several examples of this convention in the data for **`Samantha`** in Figure 1.

Possible extensions

There are a lot of things that you could do to make this program more interesting. Here are just a few possibilities:

- *Add features to the display.* The current display contains only lines and labels, but could easily be extended to make it more readable. You could, for example, put a dot at each of the data points on the graph. Even better, however, would be to choose different symbols for each line so that the data would be easily distinguishable even in a black-and-white copy. For example, you could use little circles for the first entry, squares for the second, triangles for the third, diamonds for the fourth, and so on. You might also figure out what the top rank for a name is over the years and set the label for that data point in boldface.

- *Allow deletion as well as addition.* Because the screen quickly becomes cluttered as you graph lots of names, it would be convenient if there were some way to delete entries individually, as opposed to clearing the entire display and then adding back the ones you wanted. The obvious strategy would be to add a "Delete" button that eliminated the entry corresponding to the value in the "Name" box. That approach, however, has a minor drawback given the design so far. If you added a bunch of entries to the graph and then deleted the early ones, the colors of the later entries would shift, which might prove disconcerting. Can you redesign the color-selection strategy so that displayed entries retain their color even if other entries are removed?
- *Try to minimize the overprinting problem.* If the popularity of a name is improving slowly, the graph for that name will cross the label for that point making it harder to read. You could reduce this problem by positioning the label more intelligently. If a name were increasing in popularity, you could display the label below the point; conversely, for names that are falling in popularity, you could place the label above the point. An even more challenging task is to try to reduce the problem of having labels for different names collide, as they do for **`Sam`** and **`Samantha`** in Figure 1.
- *Adjust the font size as the application size changes.* One of the wonderful features of this program is that it redraws itself to fill the available space if you change the size of the window. If you make it too small, however, the labels run together and become unreadable. You could eliminate this problem by choosing a font size that allows each label to fit in the space available.
- *Rewrite the application to use the model/view/controller pattern.* The NameSurfer prorgam turns out to be an ideal application for the model/view/controller pattern described in Chapter 14. If you finish the standard assignment, you might try to reimplement it so that it maintains a separate model that can support multiple viewers. If you made this change, you could, for example, add a "Table" button to the application that would add a second viewer to the application, which would display the results in tabular rather than graphical form.