Project 4

Computer Science 2334
Spring 2009

User Request:

"Create a Contact Information Management System with Import/Export and Load/Save Features and an Advanced Graphical User Interface including a Map."

Milestones:

1. Create a contact information management system to store contact information in an electronic address book, as done in Project 3, and add to that address book information on the global coordinates (latitude and longitude) of the addresses it contains. *5 points*
2. Create an MVC model as exemplified by the **CleanMVCCircleModel** class as discussed in lectures. The class for this model will be called "**ContactModel**." This model will contain (1) the variables and methods needed to keep track of the address book itself (including the global coordinates, as specified under milestone 1), and (2) the variables and methods necessary to keep track of the views of the address book. *10 points*
3. Create an MVC view associated with the contact information in the model to display the list of contacts. The class for this view will be called "**ContactView**." This view will be used in conjunction with the "focus" button (see below) to determine a subset of contact information to display to the user. *10 points*
4. Create an MVC view associated with *a subset of* the contact information in the model to display *a sublist* of contacts. The class for this view will be called "**FocusView**." This view will be used in conjunction with the "map" button (see below) to display a map of some (or all) of the contacts to the user. *10 points*
5. Create a *pseudo* MVC view associated with a subset of the contact information in the model to display a *map* of contacts. The class for this view will be called "**MapView**." *10 points*
6. Implement a simple Graphical User Interface "control panel" using Swing that allows for selecting among the program's functions (load, save, import, export, view, focus, and map). The control panel will use buttons for the functions listed plus a text field in a dialog to get a range for the focus. *5 points*
7. Use a **JFileChooser** dialog to allow the user to choose the file used when loading, saving, importing, or exporting the dictionary. *5 points*
8. Create an MVC controller called "**ContactController**" associated with the model. When the user clicks "load," "save," "import," or "export," the controller will tell the model to load, save, import, or export itself, respectively. When the user clicks "view," "focus," or "map," the controller will tell the corresponding view to display itself. *15 points*

- ► Develop and use a proper design. *15 points*
- ► Use proper documentation and formatting. *15 points*

Description:

An important skill in software development is extending the work you have done previously. For this project you will rework Project 3, using contact information for colleges and universities rather than for people, and add a graphical user interface organized around the model, view, controller (MVC) paradigm. The MVC paradigm gives us a way to organize code involving graphical data displays and/or user interfaces, particularly GUIs. Your application will graphically display contact information in lists and a map. In particular, you will create a single model to hold the data, three views to display various aspects of the data, and one controller to moderate between user gestures and the model and views. For this program you may reuse some of the classes that you developed for your previous projects, although you are not required to do so. Note that much of the code you write for this program could be reused in more complex applications, such as a an employee database or a social networking website.

Model:

You will create a model class called "**ContactModel**." Models in the version of the MVC paradigm shown in the CleanMVCCircle example from our lectures contain data and methods for the application objects being modeled (contacts, in this case) as well as data and methods to allow the model to interact with views. Your model class will follow this version of the MVC paradigm.

For the application objects, you will create an address book, similar to the one from Project 3. This address book will additionally contain information on the global coordinates (latitude and longitude) of the contacts.

The names, addresses, and phone numbers for contact objects will originally come from a text file that the model will read in and parse, known as *importing*. For each imported postal address, the model will add a set of global coordinates to the corresponding contact object by querying Google Maps.

The model will also have the ability to export its application data to a text file. This process will be known as *exporting*. For exporting, the model will only write out the basic information (name, street address or PO box, city, state, zip, and phone numbers) – it will *not* write out the global coordinates – and will use the same format as the imported text file.

The model will also have the ability to *load* and *save* contact information in a "native" file format, rather than a text file. Loading and saving will be object I/O, so they will load and save *all* information (name, street address or PO box, city, state, zip, phone numbers, *and* global coordinates) for each contact.

The other basic operation that your model should be able to perform is to do a distance-based search. Given a reference to one contact object and a distance, **ContactModel** should return a list of all of its contacts that are within the range specified by the distance.

To interact with views, the **ContactModel** class will have variables and methods akin to those from the **CleanMVCCircleModel** class from our lectures. In particular, when contacts are read into the address book in the model, the **ContactView** and **FocusView** objects should be notified, and when the focus is changed, the **FocusView** objects should be notified. (Please note that **MapView** objects do not need to be notified when the **ContactModel** changes. For this reason, we are calling **MapView** a "pseudo-view." If you wish to update the **MapView** when the model changes, you may do this for extra credit.)

Views:

Producing views of information can be very useful to users. Therefore your program will create and maintain three views of the data: one list of all the contacts in the address book called "**ContactView**," one sublist of just those contacts on which we are focusing at this time called "**FocusView**," and one "pseudo-view" showing a map of the contacts on which we are focusing called "**MapView**." Each view

will be given its own window, which should be a **JFrame** for **ContactView** and **FocusView** but an external application (a web browser) for **MapView**.

Within the **JFrame** for **ContactView** and for **FocusView** you should place a scrolling list of contacts. The list for **ContactView** should allow for selections by the user. In particular, when the user clicks on "focus" on the control panel, the current selection in the **ContactView** should be used to determine the sublist of contacts to be used for the **FocusView** and the **MapView**. You should ensure that only one entry in the **ContactView** can be highlighted at any given time. Based on distance to this entry selected from **ContactView**, a subset of nearby contacts will be selected for display in **FocusView**.

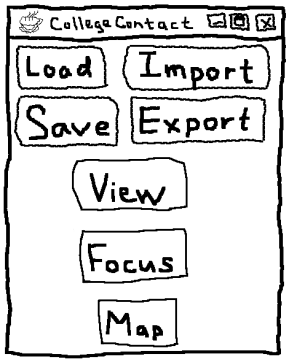
As explained above, **MapView** objects do not need to be notified when the **ContactModel** changes, which is why we are calling **MapView** a "pseudo-view." To implement the functionality, you will have your code interact with the Google Maps API to provide information to the web browser.

Controller:

You will create a controller class called "**ContactController**" to handle the task of asking the model to update itself in response to user input and selecting views. In particular, the controller will be responsible for the following:

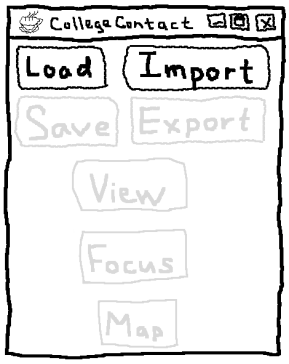
1. When the user asks to load/import (or save/export) an address book, the **ContactController** will tell the model to read in (write out) a file specified by the user through a **JFileChooser** dialog.
2. When the user asks to see a view, the **ContactController** will tell the corresponding view to display itself.

GUI:



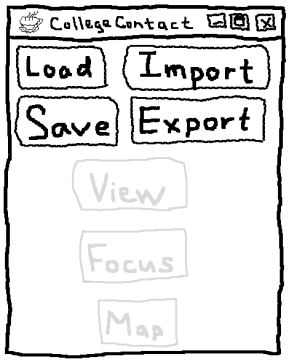
The basic Graphical User Interface for Project 4 will have seven buttons plus one text box. The seven buttons will be labeled "load," "save," "import," "export," "view," "focus," and "map." The text box will be for entering the range to be used for selecting a sublist of contacts on which to focus. The text box will be a pop-up dialog box. A rough sketch of the basic GUI is shown in the figure to the right. (The figure is intentionally rough so that you will not attempt to copy it exactly.)

This main window will be augmented by additional windows that will pop up to help with file selection and to display the three views. In particular, a **JFileChooser** window will pop up to help with file selection for loading, saving, importing, or exporting; a **JFrame** will pop up to display **ContactView** and **FocusView**; an external application (a web browser) will pop up for **MapView**.



Loading/Saving and Importing/Exporting:

When your program is run initially, it should be clear from looking at the GUI that only load and import are possible. To make this clear, the save, export, view, focus, and map buttons will be displayed differently (e.g., with the letters "grayed out"). See, for example, the figure at left.



When the load or import button is selected, a **JFileChooser** window will pop up to help with file selection. Once an address book has been loaded or imported, the other two I/O buttons (save and export) will become active, as shown at right. These will provide the user with the two corresponding options: (1) The user may choose to

save the address book, using object output. If the user makes this choice, an appropriate dialog will pop up to allow the user to specify a file name and to browse directories to determine where to save the file. (2) The user may choose to export the address book using text output. Again, if the user makes this choice, an appropriate dialog will pop up to handle file name and directory selection. Additionally, once an address book has been loaded or imported, the view button will become active.

Displaying Views:

When the view button is clicked by the user, the **ContactView** will be displayed. If the user selects one of the contacts in the **ContactView**, then the focus button will become active.

If the user clicks on the focus button while a contact is selected in the **ContactView**, then a dialog box will pop up and prompt the user for a distance from the selected contact within which to focus the system. Once a positive value is entered into the dialog, the **FocusView** will be displayed. The contents of **FocusView** will be all of the contacts from the address book that are within the specified range of the selected contact.

Once the **FocusView** is visible, the map button will become active. If the user clicks on the map button, the **MapView** will appear, showing all of the contacts from the **FocusView** in a Google map within an external web browser.

Each time the address book contents are changed (a new address book is loaded or imported), your program will update the **ContactView**, hide the **FocusView** if it is currently visible, and deactivate the focus and map button if they are currently active.

Implementation Issues:

Time Management:

This is the largest project we've had so far so make sure to start early and budget your time well. Once you have a good design, you can write a part at a time and test it before moving on to the next part. Don't expect to be able to finish the project if you put it off until the last minute; on the other hand, if you use your time well, you should have plenty of it.

Google Maps and the Google Maps API:

Background:

The Google Maps Application Programming Interface (API) uses JavaScript to create interactive maps within graphical web browser windows. (See: `http://www.google.com/apis/maps/index.html` and related pages.) You are undoubtedly already familiar with at least one web browser – how else would you have accessed this document from the class web pages? Most web browsers (e.g., Firefox, Opera, Epiphany, Konqueror, OmniWeb, Safari, etc.) are graphical, though some (e.g., Lynx) are text-based. For this assignment, you will need to use a graphical browser that is capable of processing JavaScript to display the map and have the data plotted on it.

JavaScript is a programming language used primarily for client-side scripting on the web. This means that a web browser will load a piece of JavaScript source code – typically from a web server, though in this assignment from a file – which it can then interpret and run. Because the JavaScript is normally coming from a web server to your browser (which is said to be the client which the server is serving), we call this "client-side." Because it is the source code that is loaded by the browser, we call it "scripting."

Note that this is different than the way browsers handle Java. With Java, the source code is compiled

into bytecode before being handed to the browser. The browser then uses a Java Runtime Environment as a virtual machine to run the Java bytecode.

Java and JavaScript share some syntax but are not closely-related languages. They are both used in browsers to add dynamic content, which is why Sun (which developed Java) and Netscape (which developed JavaScript) saw a marketing opportunity and agreed on the similar names.

Description:

To interact with Google Maps, we'll do two things.

First, to query Google Maps for global coordinates for our contacts, we'll create a query string within Java, open a direct URL connection to http://maps.google.com, send the query string to Google Maps as part of the URL, and read the response sent back by Google. We will then parse the received response and place the latitude and longitude values into our contact objects. A template for this code has been provided to you.

Second, to use the Google Maps API to display maps in a web browser, we'll embed our JavaScript in a small HTML wrapper. This is because the web browser is expecting the file it loads to be a web page. Inside the HTML will be information to tell the browser that we are using JavaScript ("`<script ... >`"), to fetch some base JavaScript from Google Maps (`src='http://maps.google.com/maps?file=api&amp;v=2'`), and to implement our own JavaScript functions. Our JavaScript will tell the browser to create a new map to go in the canvas, create a new center point for the map and center the map there, and to put a marker at the latitude/longitude coordinates for each contact location.

A template for this JavaScript has been provided to you. In this code, the JavaScript is a single long string called "`content`" that will be written to a file, which the browser will then open. When the browser opens the file, it will run the JavaScript (assuming the browser is JavaScript capable and has JavaScript processing turned on).

Also provided are simple examples for how to substitute into the JavaScript `content` string and how to start the browser. The substitution examples are rather like stub-code in that they allow you to see how parts of the code will function but do not implement all the functionality that you'll want in your code. In order to complete your assignment, your will need to create methods that can substitute into the `content` string the actual data selected by the user from the GUI you are creating. For the center point, you may use the latitude/longitude coordinates of the contact selected in **ContactView**.

Due Dates and Notes:

Due Dates:

Your revised design and detailed Javadoc documentation are due on **Thursday, April 9th**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your revised UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation, and a hardcopy of the stubbed source code at the **beginning of lab on Thursday, April 9th**.

The final version of the project is due on **Thursday, April 23rd**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your final UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation, and a hardcopy of the source code at the **beginning of lab on Thursday, April 23rd**.

Sources:

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

Group Work:

As noted in the syllabus, you are required to work on this programming assignment in a group of at least two people. It is your responsibility to find other group members and work with them. The group should turn in only one (1) hard copy and one (1) electronic copy of the assignment. Both the electronic and hard copies should contain the names and student ID numbers of **all** group members. If your group composition changes during the course of working on this assignment (for example, a group of five splits into a group of two and a separate group of three), this must be clearly indicated in your write-up, including the names and student ID numbers of everyone involved and details of when the change occurred and who accomplished what before and after the change.

Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project. For example, if you have three functions in your program and one function was written by group member one, the second was written by group member two, and the third was written jointly and equally by group members three and four, both your write-up and the comments in your code must clearly indicate this division of labor. Giving improper credit to group members is academic misconduct and grounds for penalties in accordance with school policies.

Extra Credit:

There are many possibilities for extra credit on the mapping side of the assignment. The key is to make them useful, explain why they are useful, and implement them well. For example, you could associate additional data relating to each point with its marker so that it appears when the user "mouses over" the point. How many points you may get for any addition depends on how useful it is and how well it is explained and implemented.