Project #4

Computer Science 2334
Spring 2012

User Request:

"Create a Presidential Nomination Information System with a Graphical User Interface."

Milestones:

1. Create appropriate classes, complete with variables and methods, to handle the application data on Presidential nominations as described below under model. *15 points*
2. Add a class, complete with data and methods, to the model to allow the model to correctly interact with the controller and the views. *10 points*
3. Create appropriate classes, complete with variables and methods, for the views described below. *25 points*
4. Create an appropriate class, complete with variables and methods, for the controller as described below. *25 points*

- ► Develop and use a proper design. *15 points*
- ► Use proper documentation and formatting. *10 points*

Description:

An important skill in software design is extending the work you have done previously. For this project you will rework Project #3 in order to handle information pertaining to the US Presidential nomination and allow users to view and manipulate that information graphically. This project will be organized around the model, view, controller (MVC) design pattern which gives us a way to organize code involving graphical user interfaces (GUIs). In particular, you will create a single model to hold the data, several views to display and manipulate different 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.

Model:

You will create a model class called "**NominationModel**." Models in the version of the MVC design pattern we have used in class contain data and methods for the application objects being modeled (which are all related to US Presidential nomination data in this project) as well as data and methods to allow the model to interact with views. Your model class will follow this version of the MVC design pattern.

For the application objects, you will create a class to represent individual candidates, a class to represent lists of candidates, a class to represent individual contests, a class to represent lists of contests, and a class to represent the overall nomination structure. The information on each individual candidate will consist of personal information (name, birthday, and birthplace) plus a list of electoral contests in which the candidate has participated. The information on each list of candidates will consist of party affiliation plus references to the candidates themselves. There will be one candidate list for each party. (Note that we have been focusing on the Democratic and Republican parties in our examples but that your program should be general enough to allow for more or fewer parties.) The information on each contest will

consist of a two letter postal abbreviation for the place (usually a state) in which the contest occurred and references to the candidates who participated in that contest along with the votes and delegates each candidate received. Please take note of this circular relationship – each candidate object will refer to a list of contests in which that candidate participated and each contest will refer to each of candidates who participated in that contest. The information for lists of contests will consist of party affiliation in addition to references to the contests themselves. (Again, we have been focusing on the Democratic and Republican parties in our examples but that your program should be general enough to allow for more or fewer parties.) The information for the overall nomination structure will consist of references to all of the candidate lists (one list per party), references to all of the contest lists (one list per party), and the year (such as "2012").

The data for this model will enter the system through three alternate ways: (1) It may be read in from a text file, which we will refer to as *importing* the data. (2) It may be read in using object I/O, which will will refer to as *loading* the data. (3) It may be entered by a person using the input views described below, which we will refer as *entering* the data.

To interact with views, the **NominationModel** class will have variables and methods akin to those from the examples we have seen in MVC lectures and labs. In particular, when new information is added to the model by importing, loading, or entering new data, any relevant display views should be notified so that they may update themselves to reflect the new data.

Views:

Producing views of information can be very useful to users. Therefore your program will create and maintain several views of the data, as described below.

Selection View:

The *Selection View* will be displayed as soon as the program is started. There will be a title in the top bar that says "<Year> <Party> Presidential Nomination" where <Year> is replaced by the year of the nomination (once it is specified) and <Party> is replaced by the party for which this nomination is taking place (once it is specified). For example, the top bar might say "2012 Republican Presidential Nomination" at some point). Note that, when the program is started and no data has yet been loaded or entered, the top bar will initially say "Presidential Nomination" only, with no year or party). In addition, there will be a menu bar with a file menu, and, in the content pane of the window, two vertical lists displayed side by side each with its own title above it and set of buttons below it. The list on the left will be titled "Candidates" and the list on the right will be titled "Contests." The buttons below the candidates list will be "New Candidate," "Delete Selected Candidate(s)," and "Delete All Candidates." The buttons below the contest list will be "New Contest," "Delete Selected Contest(s)," "Delete All Contests," "Pie Chart," and "Bar Chart." Initially, both lists will be empty and all buttons will be grayed out and inactive. The lists will be kept in sync with the underlying model information on candidates and contests, respectively, and the buttons will become active and inactive as appropriate. (That is, once there is at least one candidate, the Delete All Candidates button would be appropriate and should become active, but the Delete Selected Contest(s) , Delete All Contests, Pie Chart, and Bar Chart buttons should not be active unless the contest list contains at least one entry. Similarly, once there is at least one contest in the contest list, the Delete All Contests button would be appropriate and should become active. The Delete Selected Candidate(s), Pie Chart, and Bar Chart buttons should only become active if there is at least one candidate selected on the candidate list. And so on.) Similarly, the data menu and some menu items in the file menu will initially be grayed out and inactive.

The file menu will have entries for "New Nomination," "Load Nomination," "Save Nomination,"

"Import Nomination," and "Export Nomination." Initially, only New Nomination, Load Nomination, and Import Nomination will be active. Save Nomination and Export Nomination will be grayed out and inactive until a new nomination has been created using New Nomination or an existing one has been read in using Load Nomination or Import Nomination. As with the buttons, the menu items should only be active when executing the action would be appropriate. (For example, it makes no sense for the user to save or export a nomination if one has not been created, loaded, or imported.)

If the user selects New Nomination and there is no unsaved data in the system (i.e., this is the first time since your application was started that the user has selected New Nomination and the user has not loaded or imported an existing Nomination), your program will pop up a dialog box asking the user for a party and a year and initialize the Nomination object to have those values. If, on the other hand, there is data in the system that has not yet been saved, it will pop up a dialog box asking the user whether that data should be saved, exported, or discarded. If the user chooses to save the data, your program will save it using object I/O. If the user chooses to export the data, your program will export it to a text file. If the user chooses to discard the data, the Nomination object should be cleared. Once there is no more unsaved nomination data in the system, your program will proceed to the dialog box asking the user for a party and a year and initialize the Nomination object to have those values, as described previously.

If the user chooses Load Nomination or Import Nomination and there is unsaved data, your program will pop up a dialog box asking the user whether that data should be saved, exported, or discarded, as it did for the New Nomination option. The actions taken by your program in response to this save/export/discard dialog box should be the same as those for the New Nomination save/export/discard dialog. Once there is no unsaved data in your system, your program will present the user with a file picker and allow him or her to select an appropriate file to read in via object input or text input, as appropriate.

Once a new Nomination has been created or an existing one read in from a file, the Save Nomination and Export Nomination menu items will become active. If the user chooses Save Nomination or Export Nomination, your program will present the user with a file picker and allow him or her to designate an appropriate file for writing via object output or text output, as appropriate.

Also once a new Nomination has been created or an existing one read in from a file, the Add Candidate button under the candidate list will become active. However, the Add Contest button will not become active until at least one candidate is in the model. Additionally, if a Nomination is read in from a file and it contains candidates, the candidate list will be populated with candidates from the model and the Delete All Candidates buttons will become active, and similarly for the contests.

If the user selects Add Candidate, a dialog will pop up to ask the user for the candidate's personal information. The controller will take the information entered into this dialog and pass it along to the model, asking the model to add the candidate. However, note that the model will not allow duplicate candidate names to be added and the user should be notified by the view if any such data entry error is made. Once there is at least one candidate in the model, the Add Contest button will become active.

If the user selects Delete Selected Candidate(s), then a dialog will pop up to confirm that the user wants to delete the selected candidate(s) and its (their) data.

If the user selects Delete All Candidates, then a dialog will pop up to confirm that the user wants to delete all candidates from the Nomination.

The contest list and buttons for adding and deleting contests will behave similarly to the candidate list and buttons with respect to their data with the exceptions that (1) there must be at least one candidate in the system before contests may be added to the system, (2) when entering contest data the user will select from a list of available candidates in the system, (3) the user will be prompted to enter data (votes

and delegates) for each selected candidate, and (4) once the first contest has been added to the system, the chart buttons will become active.

If the user selects the Pie Chart button, a Pie Chart View (described below) will appear. If the user selects the Bar Chart button, a Bar Chart View (described below) will appear.

Pie Chart View:

If the user selects the Pie Chart button, your program will open up a Pie Chart View for the contest(s) selected in the contest list. The Pie Chart View have a label across the top that displays the postal abbreviations of all the selected contests. Below that will be a pie chart that shows the percentage of the votes for each candidate in those contests. Each slice of the pie will be easily distinguished from every other slice by color(s) and/or pattern(s). The slices of the pie either need to be labeled individually or there needs to be a key that shows which color pattern corresponds to each candidate. The labels need to be click-able so that the user may bring up a Candidate View (described below) for a given candidate by clicking on that candidate's label.

Bar Chart View:

If the user selects the Bar Chart button, your program will open up a Bar Chart View for the contest(s) selected in the contest list. The Bar Chart View have a label across the top that displays the postal abbreviations of all the selected contests. Below that will be a bar chart that shows the count of the delegates for each candidate in those contests. Each bar in the chart will be easily distinguished from every other bar by color(s) and/or pattern(s). The bars in the chart either need to be labeled individually or there needs to be a key that shows which color pattern corresponds to each candidate. The labels need to be click-able so that the user may bring up a Candidate View (described below) for a given candidate by clicking on that candidate's label.

Candidate View:

If the user clicks on a candidate label in either a Pie Chart View or a Bar Chart View, the program will open up a Candidate View for that candidate. The Candidate View will present the same information in the same format as the Screen Output from Project #3, except that it will be in a window rather than being displayed to the console.

Notes on Views:

Note that there will be one Selection View for your program. It will be opened when your program runs. When the user closes this view, your program should exit. In contrast, there may be many Pie Chart Views and/or Bar Chart Views and/or Candidate Views open at any given time. Each time the user selects the Pie Chart button, a new Pie Chart View should open for the contest(s) currently selected in the contest list, unless there is already a Pie Chart View open for that (those) contest(s). Similarly, for bar charts and Candidate Views. Each Pie Chart View, Bar Chart View, and Candidate View can be closed independently by the user by closing the window in which it resides. Additionally, if a contest with which a Pie Chart View or Bar Chart View is associated is deleted, the view should automatically close itself. Similarly, if a candidate with which a Candidate View is associated is deleted, the view should automatically close itself.

Controller

Besides at least one model and at least one view, every GUI-based program using the MVC design pattern needs to have at least one controller. The controller is responsible for connecting the model(s) to

the view(s) so that appropriate actions are taken in response to user gestures and that appropriate representations of data are presented to users.

For this project, the appropriate actions for user gestures and appropriate representations of data are described above, under the individual views. The controller that you create for this project, then, will need to ensure that the actions are taken and views updated as described above. Call this controller **NominationController**.

Text Input and Output

The format of the text file containing data will be the same as for Project #3. This format should be used for both importing and exporting data.

How to Complete this Project:

1 Create figures, on engineering paper or using drawing software, to show approximately what each view will contain. These figures do not need to exactly match the appearance of the final windows but should contain all major components and show their basic layout.

2 During the lab session and in the week following, you should work with your partner(s) to determine the classes, variables, and methods needed for this project and their relationship to one another. This will be your preliminary design for your software.

2.1 Make a list of the nouns you find in the project description that relate to items of interest to the "customer." Mark these nouns as either more important or less important. More important nouns describing the items of interest to the "customer" should probably be incorporated into your project as classes and objects of those classes. Less important nouns should probably be incorporated as variables of the classes/objects just described. This list will be turned in with your preliminary and final designs long with your other design documents.

2.2 Make a list of the verbs you find in the project description that relate to items of interest to the "customer." Verbs describing behaviors of the desired objects and the systems as a whole should probably be incorporated into your project as methods. This list will be turned in with your preliminary and final designs long with your other design documents.

2.3 Be sure to use UML class diagrams as tools to help you with the design process.

3 Once you have completed your UML design, create Java "stub code" for the classes and methods specified in your design. Stub code is the translation of UML class diagrams into code. It will contain code elements for class, variable, and method names; relationships between classes such as inheritance, composition, and aggregation as appropriate; variable types; and method signatures. Stub code does not, however, contain method bodies. Because we are using object-oriented design, in which the inner workings of one class are largely irrelevant to the classes with which it interfaces (that is, we are using encapsulation), we do not need to complete the implementation any class until the design is complete.

4 Add comments to your stubbed code as specified in the documentation requirements posted on the class website. Run your commented stubbed code through Javadoc as described in the Lab #2 slides. This will create a set of HTML files in a directory named "docs" under your project directory.

5 At the end of the first week (see ***Due Dates and Notes***, below), you will turn in your preliminary design documents **including your view figures**, which the TA will grade and return to you with helpful feedback on your preliminary design. Please note: You are encouraged to work with the instructor and the TAs during their office hours during the design week to get feedback throughout the design process as needed.

Final Design and Completed Project

6 Using feedback from the instructor and TAs as well as your own (continually improving) understanding of OO design, revise your preliminary UML design.

7 Make corresponding changes to your stub code, including its comments.

8 Implement the design you have developed by coding each method you have defined. A good approach to the implementation of your project is to follow the project's milestones in the order they have been supplied. If you find that your design does not allow for the implementation of all methods, repeat steps 5 and 6.

9 Test your program and fix any bugs.

10 Once you have completed the project and are ready to submit it for grading, create a new set of Javadoc files using Eclipse and inspect them to make sure your final design is properly documented in your source code.

11 Submit all parts of your completed project. (See below for due dates and requirements regarding submission of paper and electronic copies of project components.)

Extra Credit Features:

You may extend this project with more features for an extra 5 points of credit. For example, you could think of ways to present the user with helpful information about the program, such as a context-dependent help system.

To receive the full five points of extra credit, your extended features must be novel (unique) and must involve effort in the design of the extra features and their integration into the project and the actual coding of the features. Also, you must indicate on your final UML design which portions of the design support the extra feature(s); and you must include a write-up of the feature(s) in your milestones.txt file. The write-up must indicate what each feature is, how it works, how it is unique, and the write-up must cite any outside resources used.

Due Dates and Notes:

An electronic copy of your revised design including **view figures**, Noun/Verb List, UML, stub code, and detailed Javadoc is due on **Friday, April 6th**. Submit the project archive following the steps given in the submission instructions **by 5:00pm**. Submit your **view figures on *engineering paper* or hardcopies made using drawing software**, Noun/Verb List, revised UML design on *engineering paper* or a hardcopy made using UML layout software, a hardcopy of your stub source code, and a hardcopy of your cover page **by 5:00pm** on **Friday, April 6th**.

An electronic copy of the final version of the project is due on **Thursday, April 19th**. Submit the project archive following the steps given in the submission instructions **by 10:00am**. Submit your **final view figures on *engineering paper* or hardcopies made using drawing software**, final UML design on *engineering paper* or a hardcopy made using UML layout software, a hardcopy of the cover page for your project, and a hardcopy of the milestones.txt file at the **beginning of lab on Thursday, April 19th**.

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.

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 on the cover sheet. 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 cover sheet, 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, your cover sheet 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.

When zipping your project (or design) for submission, be sure to follow the instructions carefully. In particular, *before* zipping the project be sure to

- **place additional files (such as UML diagrams, cover sheets, and milestones files) within the "docs" directory inside your Eclipse folder for the given project and be sure that Eclipse sees these files (look in the Package Explorer and hit Refresh if necessary), and**

rename the project folder to the 4x4 of the team member submitting the project. Note that renaming the project folder to your 4x4 *before* zipping is not the same thing as naming the zip file with your 4x4. The latter is fine; the former is *mandatory*.