Project 4

Computer Science 2334
Spring 2017

This project is optionally group work. See Notes at end.

User Request:

"Create a sortable and searchable data system for news stories and those the stories are about, that uses text and binary input and output and ***a Graphical User Interface****."*

Milestones:

1. Create appropriate classes, complete with data fields and methods, to handle application data on news stories and those the stories are about as described below under Model. *15 points*
2. Add a class, complete with data fields 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 data fields and methods, for the views described below. *25 points*
4. Create an appropriate class, complete with data fields 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 Nooz 3.0 from Project 3 in order to handle information pertaining to news stories and those the stories are about and allow users to view and manipulate that information graphically. This graphical viewing and manipulation of data will be what distinguishes Nooz 4.0 from Nooz 3.0. Nooz 4.0 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 code 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 appropriate model classes with names ending in "**Model**." These model classes should be extensions of application data classes that (indirectly) contain data and methods for the application objects being modeled (which are all related to news stories and those the stories are about in this project) as well as data and methods to allow the model to interact with views. Your model classes will follow this version of the MVC design pattern.

For the application objects, you will have classes to represent the same kinds of things that you represented in Project 3 (news stories of various types, news makers, and lists of both). Note that while you have represented these application objects in previous projects, you may need to modify your

classes for those objects at least slightly in order to satisfy the requirements of this project. (You may, of course, modify the classes substantially for this project, if doing so would substantially improve your project.)

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.

Whether data is imported or entered, it must meet the following reality checks: (1) A story length must be positive. (2) News stories can be associated with two news makers, at most. If invalid data is found during import or entry, the user will be informed of the error and the model will not be updated to include that data.

To interact with views, the **Model** classes 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 "Nooz." In addition, there will be a menu bar with a file menu, a newsmaker menu, a news story menu, a display menu, and, in the content pane of the window, two vertical scrolling lists, side by side, one for news makers and one for news stories. Note that there will be no data in Nooz until some is imported, loaded, or entered, so initially the lists will be empty. The lists will be kept in sync with the underlying model information. Moreover the lists will be kept ordered. The newsmaker list will be ordered alphabetically. The news story list may be sorted by the user (see News Story Menu, below). These lists will be selectable, so the user can click on any or all of the news makers in the newsmaker list (once it has at least one) or any or all of the news stories in the news story list, then choose an option from one of the menus that effects objects of that type (see below).

File Menu:

The file menu will be marked "File" and have entries for "Load," "Save," "Import," and "Export." Initially, only Load and Import will be active. Save and Export will be grayed out and inactive until at least one object (a news story of some type or a news maker) has been added through the GUI or an existing one has been read in using Load or Import. In general, 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 data if none is present.)

If the user chooses Load or Import, Nooz will present the user with a file picker and allow selection of one or more files to read in via object input or text input, as appropriate. Additionally, if the user chooses Import, Nooz will ask which type of data the file contains (source codes, topic codes, subject codes, or news stories).

Once data is present in the system, the Save and Export menu items will become active. If the user chooses Save or Export, Nooz will present the user with a file picker and allow designation of one or

more files for writing via object output or text output, as appropriate.

For each inactive menu item, if the user hovers the pointer over that component, a tool tip will appear to let the user know why the button is not active. (For example, "Cannot export data; no data present.")

Newsmaker Menu and Associated Views:

The newsmaker menu will be marked "Newsmakers" and have entries for "Add Newsmaker," "Edit Newsmaker," "Delete Newsmaker," and "Delete Newsmaker List." As with the file menu, menu items should be grayed out and inactive until they are appropriate. Since only "Add Newsmaker" is appropriate until data is entered, it will be the only one initially active.

If the user selects Add Newsmaker, a *Newsmaker Entry View* will appear and provide the user an interface to allow a newsmaker to be added. This entry view will simply allow a user to enter a newsmaker name. If the newsmaker already exists in the system, the user will be give the option to replace the existing newsmaker with that name with the new newsmaker. This will have the effect of removing that newsmaker from any stories that currently refer to that newsmaker while still having the newsmaker name in the system (equivalent to deleting the newsmaker, then adding a new newsmaker with that name).

If the user selects an entry from the newsmaker scrolling list (in the content pane of the Selection View) and clicks Edit Newsmaker, then a *Newsmaker Edit View* will appear to allow the user to edit the newsmaker. This edit view will display the newsmaker's current name, a scrolling list of all the news stories currently associated with that user, and a button marked "Remove from Story." Two types of edits are possible: name and stories.

First, the user could change the newsmaker's name by editing the text of the name displayed. If the edited newsmaker name is *not* already present in the system, the newsmaker's name will be updated to the one specified by the user. If the edited newsmaker name *is* already present in the system, the user will be give the option to replace the existing newsmaker with that name with the newsmaker being edited. This will be equivalent to deleting the newsmaker that formerly had that name, then renaming the edited newsmaker to have that name.

Second, the user could remove the newsmaker from one or more of the stories on that newsmaker's list. The user would do this by selecting one or more of the stories on the newsmaker's list of news stories, then clicking the button marked "Remove from Story." The effect will be to replace the selected newsmaker with the special newsmaker "None" in each of the selected stories.

If the user selects more than one newsmaker from the newsmaker list and clicks Edit Newsmaker, then a series of edit views will appear, one for each selected newsmaker.

If the user selects one or more newsmakers on the scrolling list and clicks Delete Newsmaker, then a dialog will appear to confirm whether the user wants to delete the selected newsmaker(s). Note that if an newsmaker is deleted from the system, all references to that newsmaker should be changed to refer to "None." (Note that "None" cannot be removed from the system.)

If the user selects Delete Newsmaker List, then a dialog will appear to confirm whether the user wants to clear (delete) all of the newsmakers. As with delete, if a newsmaker is deleted from the system, all references to that newsmaker should be deleted from the system (meaning that all stories would be changed to refer to "None").

News Story Menu and Associated Views:

The news story menu will be marked "News Stories" and have entries for "Add News Story," "Edit News Story," "Sort News Stories," "Delete News Story," and "Delete All News Stories." As with the

file and newsmaker menus, menu items should be grayed out and inactive until they are appropriate. Since only "Add News Story" is appropriate until data is entered, it will be the only one initially active.

If the user selects Add News Story, a *News Story Entry View* will appear and provide the user a window that allows a news story to be added. This entry view will provide the user the ability to enter information on all aspects of a story, from its media type to its newsmakers. Some of these aspects will be chosen from a limited set of options (e.g., the media type or the part of day) whereas others should provide the user with existing options from which to choose but should also allow the user to add values not already present in the system (e.g., newsmakers) and yet others should simply allow the user to enter their own values (e.g., length). Rather than specifying all these details for you, part of your design will be to determine an appropriate composition for this window, keeping the constraints of this paragraph in mind.

If the user selects an entry from the news story scrolling list (in the content pane of the Selection View) and clicks Edit News Story, then a *News Story Edit View* will appear to allow the user to edit the news story. This edit view will be similar to the News Story Entry View but with the current data on the story reflected in the initial window as it's displayed to the user.

If the user selects more than one news story from the news story list and clicks Edit News Story, then a series of edit views will appear, one for each selected news story.

If the user selects Sort News Stories, a submenu will allow the user to select source, topic, length, date, or subject. Whichever selection the user makes, the list of news stories will be sorted based on that feature using a stable sort.

If the user selects one or more news stories on the scrolling list and clicks Delete News Story, then a dialog will appear to confirm whether the user wants to delete the selected news story or stories.

Similarly, if the user selects Delete All News Stories, then a dialog will appear to confirm whether the user wants to all of the news stories. This will have the effect of deleting all news stories from the system but leaving the list of newsmakers intact.

Display Menu and Corresponding Display Views:

The display menu on the Selection View will be marked "Display" and have entries for "Pie Chart" and "Text."

The Pie Chart view will be very similar to the graphical output seen in Project 3 with three differences: (1) The data displayed will be based on the current selection(s) in the news maker list. For example, if the user has highlighted two newsmakers in the newsmaker list and selects Pie Chart, then the user will be presented with two pie charts (one for each newsmaker). (2) The details of the data displayed (e.g., which media types will be displayed) will be based on a popup window rather than a text conversation in the console. (3) If the data in the model is updated, the graphical output should change to match the new model data.

The Text display will be similar to the text output from Project 3 with four differences: (1) The data displayed will be displayed in its own window, rather than sent to the console, with the header line used as the window title, the bulk of the data contained in a scrolling list in the window, and the summary line displayed at the bottom of the window independent of the scrolling list. (2) The data displayed will be based on the current selection(s) in the news maker list. For example, if the user has highlighted two newsmakers in the newsmaker list and selects Text, then the user will be presented with two text displays (one for each newsmaker). (3) The details of the data displayed (e.g., the sort order) will be based on a popup window rather than a text conversation in the console. (4) If the data in the model is

updated, the graphical output should change to match the new model data.

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 Display Views open at any given time. Each time the user selects a display menu item, one or more new Display Views should open. Each Display View can be closed independently by the user by closing the window in which it resides. Additionally, if all of the data with which a particular Display View is associated are 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, then, will need to ensure that the actions are taken and views updated as described above. Call this controller **NewsController**.

Text Input and Output

The format of the text files containing data will be the same as for Project 3.

How to Complete this Project:

1 During the lab session and in the week following, you should create figures, 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. This will be a part of your preliminary design for your software.

2 During the lab session and in the week following, you should determine the classes, variables, and methods needed for this project and their relationships to one another. This will be the other part of your 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 design along 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 design along with your other design documents.

2.3 Make a list of adjectives, if any, in the project description. Adjectives often describe features of objects that could be incorporated into your project as interfaces to be instantiated by your classes. This list will be turned in with your design along with your other design documents.

2.4 Relate the nouns, verbs, and adjectives (if any), to classes/variables, methods, and interfaces (if

any) and list them. This list will be turned in with your preliminary and final designs long with your other design documents.

2.5 Next, use UML class diagrams as tools to help you establish proper relationships between your classes, variables, methods, and interfaces (if any).

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 (except for return statements for methods that return values or object references – these should return placeholders such as `null`). 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 design documents **including your view figures**, which the TA will grade and return to you with helpful feedback on your 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 After the design due date, you will be provided with a UML design that you must follow for your completed project.

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

8 Implement the design you have developed by coding each method.

9 Test each unit as it is implemented and fix any bugs.

10 Test the overall program and fix any bugs.

11 Submit all parts of your completed project (see ***Due Dates and Notes***, below).

Due Dates and Notes:

Your design (**view figures**; list of nouns, verbs, and adjectives and their mappings; UML; stub code; and detailed Javadoc documentation) is due on **Monday, 3 April 2017**. Submit the project archive following the steps given in the submission instructions **by 11:59pm**. **Submit your view figures by photographing or scanning your hand drawn figures or exporting to png or pdf figures made using drawing software, then place them in the doc directory before creating your project archive for submission.**

The final version of your project including final design (UML, Javadoc, unit tests) and final implementation is due on **Monday, 17 April 2017**. Submit the project archive following the steps given in the submission instructions **by 11:59pm**.

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.

You are allowed to work on this programming assignment in a team. The team should turn in only one (1) copy of the assignment to D2L and one to Mimir. Both copies should contain the names and student ID numbers of **all** team members in the Javadoc comments at the top of the driver class. In addition, the Javadoc comments for individual classes or methods (as appropriate) should indicate who worked on each portion of the code, in what capacity, and to what degree. 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 described in the comments at the top of the driver class, 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 team member is required to contribute equally to each project, as far as is possible. You must thoroughly document which team members were involved in each part of the project. Giving improper credit to team members is academic misconduct and grounds for penalties in accordance with school policies.