Project #4

Computer Science 2334
Fall 2013

User Request:

*"Create a scholar and scholarly publication data system **with a Graphical User Interface**."*

Milestones:

1.	Create appropriate classes, complete with data fields and methods, to handle application data on scholars and their publications 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 Project #3 in order to handle information pertaining to scholars and their publications 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 "ScholarshipModel." 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 scholars and their publications 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 scholars, a class to represent scholarly papers (conference papers or journal articles), a class to represent individual conferences, a class to represent individual journals, and a class to represent the overall scholarship structure. Note that you have represented many of these application objects in previous projects. However, many of you have not previously represented conferences or journals – only the papers published within them. Moreover, this project requires representing additional information about several of the object types you have previously represented and different ways of organizing some information, so read these descriptions carefully. For example, we previously only considered a scholar's work as an author, so we generally referred to them as ‘authors.’ However, here we will also consider other aspects of our scholars, and will refer to them more generally as ‘scholars’ and only use the term ‘authors’ when making reference to their authorship of papers.

The information on each individual scholar will consist of the scholar's name (broken into primary and secondary names as in previous projects), institutional affiliation(s), and research area(s), plus a collection of that scholar's publications and collections of that scholar's scholarly organizational efforts on behalf of journals and conferences (detailed below).

The information on each conference paper will consist of a collection of authors (references to scholars), the title of the paper, a reference to the conference in which the paper was published, the page numbers of the paper, and an optional digital object identifier.

The information on each journal article will consist of a collection of authors (references to scholars), the title of the article, a reference to issue in which the article was published, the page numbers of the article, and an optional digital object identifier.

The information for each conference will consist of the name of the organization sponsoring the conference and a collection of meetings of that conference. Each meeting will, in turn, consist of the month and year in which the meeting occurred, the location of the meeting (see below), a collection of the conference program chairs (references to scholars), a collection of the conference program committee members (also references to scholars), and a collection of the papers published within the conference.

The information for each journal will consist of the name of the organization that publishes the journal, the location where the journal is published, and a collection of volumes of that journal. Each volume will, in turn, consist of a collection of issues in that volume. Each issue will, in turn, consist of the month and year in which the issue was published, a collection of editors (references to scholars), a collection of reviewers (also references to scholars), and a collection of the articles published within that issue of the journal.

To understand the similarities between conferences and journals, it may be helpful to know that program chairs for conferences and editors for journals serve similar roles – they solicit papers for possible inclusion in their publication, send submitted papers out for review, and make final decisions about which papers to publish based on the reviews they receive. Likewise, the program committee members for conferences and the reviewers for journals serve similar roles – they review the papers assigned to them and make recommendations for inclusion or exclusion of the papers they review.

A location for a conference or a journal consists of a city name, an optional state or province name, and a country name.

The information for the overall scholarship structure will consist of references all of the scholars, all of the journals, and all of the conferences.

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 **ScholarshipModel** 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 “ScholarPub.” In addition, there will be a menu bar with a file menu, a plot menu, and, in the content pane of the window, three 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 “Scholars,” the list in the middle will be titled “Serials,” and the list on the right will be titled “Papers.” The buttons below the scholars list will be “Add Scholar,” “Delete Selected Scholar(s),” and “Delete All Scholars.” The buttons below the serials list will be “Add Serial,” “Delete Selected Serial(s),” and “Delete All Serials.” The buttons below the papers list will be “Add Paper,” “Delete Selected Papers(s),” and “Delete All Papers.”

Initially, all lists will be empty and most buttons will be grayed out and inactive. The only initially active button will be Add Scholar. The lists will be kept in sync with the underlying model information on scholars, serials, and papers, respectively, and the buttons will become active and inactive as appropriate. For example, once there is at least one scholar, the Delete All Scholar(s) button would be appropriate and should become active, but the Delete Selected Serials(s), Delete All Serials, Delete Selected Paper(s), and Delete All Papers buttons should not be active unless the serials list or the papers list (as appropriate) contains at least one entry. Similarly, some menu items in the file menu will initially be grayed out and inactive.

If the user selects Add Scholar, a dialog will pop up to ask the user for the scholar’s name, affiliation, and research areas. The controller will take the information entered into this dialog and pass it along to the model, asking the model to add the scholar. However, note that the model will not allow duplicate scholar 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 scholar in the model, the Add Serial button will become active.

If the user selects Delete Selected Scholar(s), then a dialog will pop up to confirm whether the user wants to delete the selected scholar(s) and its (their) data. Note that if a scholar is deleted from the system, that scholar should be removed from all papers and serials. If this results in an empty author list on any paper, that paper should be removed from the system. Likewise, if removing a scholar from the system results in any serial having an empty editor/program chair list or an empty reviewer/program committee list, that serial should be removed from the system.

If the user selects Delete All Scholars, then a dialog will pop up to confirm whether the user wants to delete all scholarship data. Note that if all scholars are deleted, then all lists of authors, editors, etc. will necessarily be empty, so all scholarship data will be deleted.

The serial list and buttons for adding and deleting serials will behave similarly to the scholar list and buttons with respect to their data with the exceptions that (1) there must be at least one scholar in the system before serials may be added, (2) when entering editors/program chairs or reviewers/program committee members, the user will select from a list of available scholars in the system, and (3) once the first serial has been added to the system, the Add Paper button will become active.

The paper list and buttons for adding and deleting papers will behave similarly to the serial list and buttons with respect to their data. Note that (1) there must be at least one scholar and one serial in the system before papers may be added, (2) when entering authors, the user will select from a list of available scholars in the system, and (3) once the first paper has been added to the system, the items in the plot menu will become active.

The file menu will be marked "File" and have entries for "Load Scholarship," "Save Scholarship," "Import Scholarship," and "Export Scholarship." Initially, only Load Scholarship, and Import Scholarship will be active. Save Scholarship and Export Scholarship will be grayed out and inactive until at least one scholar, serial, or paper has been added through the GUI or an existing one has been read in using Load Scholarship or Import Scholarship. 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 scholarship data if none is present.)

If the user chooses Load Scholarship or Import Scholarship 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. The actions taken by your program in response to this save/export/discard dialog box should be to save, export, or discard this data, as specified by the user. 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 the file is read in, if it contains scholars, the scholar list will be populated with scholars from the model and the Delete All Scholars button will become active, and similarly for serials and papers.

Once scholarship data is present in the system, the Save Scholarship and Export Scholarship menu items will become active. If the user chooses Save Scholarship or Export Scholarship, 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.

Display Views:

The plot menu on the Selection View will be marked "Plot" and have entries for "Type of Publication," "Publications Per Year," "Conference Papers Per Year," "Journal Articles Per Year," and "Number of Co-Authors Per Publication." If any of these menu entries is selected, the user will then be asked to select from a list of available scholars in the system. ScholarPub will then display one of five bar charts to the user, as described in Project 3. These charts will now be considered display views and must be kept in sync with the data present in the model.

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 plot menu item, a new Display View should open for the scholar selected by the user, unless there is already a Display View open for that aspect of that scholar. Each Display View can be closed independently by the user by closing the window in which it resides. Additionally, if a scholar with which a Display 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 **ScholarPubController**.

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 (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 Create unit tests using JUnit for all of the non-trivial units (methods) specified in your design. There should be at least one test per non-trivial unit and units with many possible categories of input and output should test each category. (For example, if you have a method that takes an argument of type `int` and behaves differently based on the value of that `int`, you might consider testing it with a large positive `int`, and small positive `int`, zero, a small negative `int`, and a large negative `int` as these are all likely to test different aspects of the method.)

6 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

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

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

9 Make corresponding changes to your unit tests.

10 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 7, 8, and 9.

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

12 Test the overall program and fix any bugs.

13 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.

14 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:

The electronic copy of your preliminary design (including **view figures**, Noun/Verb List, UML, stub code, detailed Javadoc documentation, and unit tests) is due on **Wednesday, October 30th**. Submit the project archive following the steps given in the submission instructions **by 10:00pm**. 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, and a hardcopy of your cover page at the **beginning of lab on Thursday, October 31st**.

The electronic copy of the final version of the project is due on **Wednesday, November 13th**. Submit the project archive following the steps given in the submission instructions **by 10:00pm**. 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, November 14th**.

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),**
- **compress all files into a .zip format. The formats .rar and .7z will no longer be accepted. Also, when submitting the initial design make sure that the UML is in one of the following formats: .png .jpg or .pdf. Custom formats such as .uml or .dia are NOT acceptable. If you are unsure how to export the file in that format, take a screen-shot of the diagram and attach that, 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*.**