MPP Design and Implementation Project

You will work in groups to design and implement a solution to the Library Problem, described below. You will create a class diagram in which class attributes and operations, associations, and inheritance relationships are shown. You will create sequence diagrams to model several use cases. You will design a usable user interface to support the required functionality of the system. And you will implement your designs in Java, using JavaFX for the user interface. For this particular project, your system will not interact with a database; to facilitate data reads and writes, object serialization will be used (as described in class and illustrated in the `library` package).

Library Problem Statement

Create the first iteration of a system that a librarian can use to check out books for library members, and that an administrator can use to add new books to the collection, create new library members, and edit library member information.

Most books may be borrowed for 21 days, but some books only for 7 days. COS

All members of the library are assigned a unique member number. First and last names, address (street, city, state, zip) and phone number of every member are also stored as member data. Properties

Books have a title, ISBN number, list of authors, and availability. Authors have first and last names, address (street, city, state, zip), phone number, credentials, and a short bio. Properties

The library has multiple copies for some popular books. Every copy of a book has its own unique copy number. (Note: A "copy" of a book is an instance of the book. For every book in a library, there is at least one copy; for popular books, there may be more than one copy. In this context, a "copy" of a book is *not* a reproduction of an original; this is a different meaning of "copy," not used here.)

Also, for each library member, the system will keep a record of his/her checkout activities in a *checkout record*. A checkout record consists of a collection of *checkout entries*. Each entry records each item checked out, the date of checkout, and the due date. The checkout record for a library member is therefore a complete record of every book that the member has ever checked out. We expect that in later phases of the library system, the checkout record will also include a record of fines for late returns and dates paid. Properties COS

In order to access the system, a librarian or administrator must login. Administrators are able to add/edit member info and add books to the collection, but they are not allowed to checkout books for a member (unless they also have Librarian access). Librarians are allowed to checkout books but not allowed to add/edit members or add books (unless they also have Administrator access). Admin Behavior

Use Cases

The system you design should support the following use cases:

1. Login

The first screen a user of the system sees is the login screen, which requests ID and password. When the Submit button is clicked, the ID is looked up in the data store. If this ID can be found, and if the password for this ID matches the password submitted, the authorization level is returned. Authorization levels are LIBRARIAN, ADMIN, and BOTH. If login is successful, UI features are made available according to the authorization level of the user.

2. Add a new library member to the system.

When an Administrator selects the option to add a new member, then he is presented with a form with fields: member id, first name, last name, street, city, state, zip, telephone number. After the data is entered and submitted, it is persisted using the persistence mechanism for this project.

3. Checkout a book (if available) for a library member.

A librarian can enter in a form a member ID and an ISBN number for a book and ask the system whether the requested item is available for checkout. If ID is not found, the system will display a message indicating this, or if the requested book is not found or if none of the copies of the book are available, the system will return a message indicating that the item is not available. If both member ID and book ID are found and a copy is available, a new checkout record entry is created, containing the copy of the requested book and the checkout date and due date. This checkout entry is then added to the member's checkout record. The copy that is checked out is marked as unavailable. The updated checkout record is displayed on the UI and is also persisted. The display of the checkout record uses a JavaFX TableView, with all cells of the table read-only.

4. Add a copy of an existing book to the library collection.

An administrator can look up a book by ISBN and add a copy to the collection. The result is then persisted.

Optional Use Cases (+1 point for doing any two of these)

1. Add a book to the library collection.

An Administrator can add a book by selecting an "add book" option. The system responds by displaying a screen with the necessary fields (ISBN, title, authors, maximum checkout length, number of copies). When the data is submitted, it is persisted.

2. Given a library member id, print (to console) the checkout record of that library member

The Librarian can search for a library member by ID, and then select an option to print the checkout record. When this is done, the checkout record appears in the console with aligned columns.

3. Determine whether a given copy of a publication is overdue, and which library member has it in his/her possession.

"Overdue" means that the due date is before today and the status of the copy is "not available". This should be implemented by providing a special screen for searching for books. The screen should make it possible to search for a book by ISBN, and the display should show, in a read-only TableView, the ISBN, book title, copy numbers of each copy, which library member (if any) has checked out a given copy, and when that copy is due back in the library.

Development Steps

1. Create the class diagram, at first *without* operations [see design workshop notes]

a. Isolate the *concepts* or *candidate classes* as described earlier in the course, by examining and filtering nouns and noun phrases in the problem statement.

b. Decide on a set of classes

c. Add appropriate attributes to the classes

d. Identify inheritance relationships

e. Create a more complete class diagram that includes the above as well as associations between your classes. Add names and/or roles to your associations as well as multiplicities. Check if there are any relationships that should be modeled with an association class. Decide if some associations should be changed to dependencies and make the changes in your class diagram.

2. To help identify operations in your classes, create one sequence diagram for each of the use cases described above. The actor in each case is either a Librarian or an Administrator who will be using the system. The actor will be interacting with the UI, so the first object the actor sends a message to will be, in each case, the UI. Each user request will be handled by an event-handler. There should be a SystemController class that organizes the steps of execution necessary to fulfill the needs of the event-handlers; the SystemController will accomplish this by delegating tasks to appropriate objects in the system and gathering the results for the event handler.

3. Use your sequence diagrams to help identify operations in your classes, and add these to your class diagram.

4. Design the user interface

a. Spend time thinking about a good way to organize the look of the UI, given the use cases that need to be supported. Can everything be done on a single screen? Should you use a menu? Aim to make it possible for the user to accomplish each use case with as few steps as possible. Plan to use a `TableView` to display (read-only) checkout records. (See the sample code in the package `lesson6.lecture.javafx.tables`.) Draw by hand sketches of the screens you decide to use.

b. Code the UI based on your sketches.

5. Data Access

a. This project will not make use of a database. Instead, all classes that need to be persisted will be stored using *object serialization*. Plan to have a separate package called `dataaccess` and a subpackage `storage`. All persisted classes (via serialization) should be placed inside this `storage` package.

b. A `main` method in `dataaccess.TestData` will load up test data to run tests on the application. This data will include at least 4 library members and 4 books (two of which have 3 copies, the rest, just 1 copy). All 4 library members should have at least 1 book checked out. Two members should each have an overdue book.

c. Each persistence operation, and each read operation, should be represented by a public method on a `DataAccessFacade` class – for instance, `saveNewMember`, `findCustomerById`. Read operations will locate the serialized objects in the `storage` package, deserialize them, perform other logic as necessary, and return the results. Save operations will store the appropriate objects in serialized form in the `storage` package.

NOTE: A data access framework, created according to the guidelines above, has been provided for you, with generic methods for storing and retrieving objects. The main functionality is in the class DataAccessFacade, which implements the DataAccess interface. You can add new methods to these as necessary.

6. Group Presentation Guidelines for the design parts of the work

- Each group will present its design and implementation of the Library System – approximately 15-20 minutes for each presentation.
- The objective of the presentation is to show how use cases are modelled in sequence diagrams (with reference to the class diagram as necessary) and then implemented starting with a user action on the UI and leading to a sequence of steps through the system to achieve the intended goal.
- Each member of your group should present some aspect of the system that you have built.
- Each major flow should be modelled in a sequence diagram and someone in your group should explain the flow and discuss how it is realized in code.
- If your group does any of the Extra Credit use cases, these should be mentioned during the presentation so that you will receive credit for them.

7. Project Submissions

- Each group must submit its project by uploading all the project files in one zipped folder, and the presentation video in another zipped folder.
- The project start-up code that you receive will contain a docs folder, which contains an instructions file and a readme_from_students file. Place the following items in the docs folder:
  - Your final class diagram – provide an image file (jpg, png, etc) rather than the mdj format provided by StarUML.
  - Your sequence diagrams – again, create image files rather than submitting mdj files.
  - In the readme_from_students file, indicate any extra credit use cases you have done and any other features of your code that we should be aware of.

8. Project Requirements

- Your code must work! the grader should be able to run the main method and find that the code works. Trying out the UI should result in successful user experience, without runtime exceptions.

- Each use case that has been implemented must have an accompanying sequence diagram. There are a minimum of four such use cases (Login, Add New Member, Add Book Copy, Checkout Book). (Note that Login was partially implemented for you, but is not complete.)

- Each of your use case implementations should be able to handle invalid user input in a reasonable way – *without* throwing a runtime exception. For instance, if a user leaves an important field blank in adding a new Library Member, the system should respond without crashing. One way to handle this is to make it so the system does nothing if input data is invalid; a better way is to arrange for the system to display an appropriate error message on the screen.

- The intended architecture must be followed. This means that
  - None of the UI classes directly accesses the data access package (all requests for data go through the SystemController)
  - The DataAccessFacade does not perform any logic other than providing access to the data store. For example, you cannot expect the DataAccessFacade to transform a list of objects into an array of objects, or to decide whether a particular book is overdue—these are activities that need to be implemented in the business layer.

- Associations are implemented correctly. This implies that bidirectional associations along with their multipicities have correct implementations, and are maintained during execution.