Assignment 3

Photos

Posted Fri, Oct 16

GUI Storyboard Due (Bitbucket) Sun, Oct 25, 11 PM

Complete Implementation Due (Bitbucket) Wed, Nov 18, 11 PM

Worth 225 points (22.5% of course grade)

For this assignment you will build a single-user photo application that allows storage and management of photos in one or more albums.

All user interaction must be implemented in Java FX, and all UIs--except for standard Java FX dialogs such as Alert and TextInputDialog--must be designed in FXML.

You may use multiple stages to show complex secondary windows, and switch multiple scenes within a stage.

You will continue working with your partner. Read the DCS Academic Integrity Policy for Programmming Assignments - you are responsible for this. In particular, note that **"All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean".**

Contents

Features

Your application must implement the following features:

Date of photo

Since we won't examine the contents of a photo file to get the date the photo was taken, we will instead use the last modification date of the photo file (as provided via the Java API to the filesystem) as a proxy. (The user interface will still refer to this as the date the photo was taken.)

To store and manipulate dates and times, you have two options:

- You can use a `java.util.Calendar` instance.
  In which case, when you set a date and time on an instance, also make sure you set milliseconds to zero, as in `cal.set(Calendar.MILLISECOND,0)`, otherwise your equality checks won't work correctly.
- Alternatively, you may use the classes in the `java.time` package.

Tags

Photos can be tagged with pretty much any attribute you think is useful to search on, or group by. Examples are location where photo was taken, and names of people in a photo, so you can search for photos by location and/or names.

From the implementation point of view, it may be useful to think of a tag as a combination of **tag name** and **tag value**, e.g. ("location","New Brunswick"), or ("person","susan"). A photo may have multiple tags (name+value pairs), but no two tags for the same photo will have the same name and value combination.

Additional details:

- You can set up some tag types beforehand for the user to pick from (e.g. location)
- Depending on the tag type, a user can either have a single value for it, or multiple values (e.g. for any photo, location can only have one value, but if there's a person tag, that can have multiple values, one per person that appears in the photo)
- A user can define their own tag type and add it to the list (so from that point on, that tag type will show up in the preset list of types the user can choose from)

Location of Photos - Stock photos and User photos

There are two sets of photos, **stock photos** that come pre-loaded with the application, and **user photos** that are loaded/imported by a user when they run the application.

- Stock photos are photos that you will keep in the application's workspace. You must have no fewer than 5 stock photos, and no more than 10.

  Create a special username called "stock" (no password, or password="stock") and store the stock photos under this user, in an album named "stock".

  Leave the photos in the application's workspace so the graders can test your application starting with your stock photos, then load other photos from their computer, see "User photos" below.

  Try to work with low/medium resolution pictures for the stock photos because they will be on Bitbucket and downloaded by the graders, and you don't want to bloat your project size.

- User photos are photos that your application can allow a user to load from their computer, so they can be housed anywhere on the user's machine. The actual photos must NOT be in your application's workspace. Instead, your application should only store the location of the photo on the user's machine. User photo information must NOT be in the released project in Bitbucket since each installation of your application on a machine will have its own set of users.

Login

- When the application starts, a user logs in with username. Password implementation is optional. It makes for a "real" scenario, but is irrelevant to the essence of the project. (There is no credit for the password feature, if you choose to implement it.)

Admin Subsystem

- There must be a special username **admin** that will put the application in an administration sub-system. The **admin** user can then do any of the following:
  - List users
  - Create a new user
  - Delete an existing user

  Note: If you elect to implement passwords for users, make "admin" the password for the **admin** user, so it's easier to grade. Otherwise we will need to ask you, or look in some README file, etc, which just turns out to be a needless hassle.

Non-admin User Subsystem

- Once the user logs in successfully, all albums and photo information for this user from a previous session (if any) are loaded from disk.

  Initially, all the albums belonging to the user should be displayed. For each album, its name, the number of photos in it, and the range of dates (earliest and latest date) on which photos were taken must be displayed. Use your discretion on how to show this additional information.

- The user can then do the following:
  - Create albums
  - Delete albums
  - Rename albums
  - Open an album. Opening an album displays all photos, with their *thumbnail* images and captions, inside that album. Once an album is open the user can do the following:
    - Add a photo
    - Remove a photo
    - Caption/recaption a photo
    - Display a photo in a separate display area. The photo display should also show its caption, its date-time of capture (see **Date of photo** below), and all its tags (see **Tags** below).
    - Add a tag to a photo
    - Delete a tag from a photo
    - Copy a photo from one album to another (multiple albums may have copies of the same photo)
    - Move a photo from one album (source) to another (the photo will be removed from the source album)
    - Go through photos in an album in sequence forward or backward, one at a time, with user interaction (manual slideshow)
  - Search for photos (Photos that match the search criteria should be displayed in a similar way to how photos in an album are displayed). Under this, you should provide the following specific features:
    1. Search for photos by a date range.
    2. Search for photos by tag type-value pairs. The following types of tag-based searches should be implemented:
       - A single tag-value pair, e.g person=sesh
       - Conjunctive combination of two tag-value pairs, e.g. person=sesh AND location=prague
       - Disjunctive combination of two tag-value pairs, e.g. person=sesh OR location=prague
       - For conjunctions and disjunctions, if a tag can have multiple values for a photo, it can appear on both arms of the conjunction/disjunction, e.g. person=andre OR person=maya, person=andre AND person=maya

       You are NOT required to do conjunctions/disjunctions on more than two tag-values pairs. In other words, you are not required to do stuff like `t1=v1 and t2=v3 and t3=v3`
  - There should be functionality to create an album containing the search results.

    As mentioned earlier (under Copy a photo from one album to another), a photo can be in multiple albums. Creating an album out of search results means copying these photos to a new album, *without deleting them from the current album(s) to which they belong*.

  Note: A single user may not have duplicate album names, but an album name may be (coincidentally) duplicated across users.

Logout

- The user (whether admin or non-admin) logs out at the end of the session. All updates made by the user are saved to disk.
- After a user logs out, the application is still running, allowing another user to log in.

Quit Application

- There should be a way for the user to quit the application **safely** at any time, bypassing the logout step. **Safely** means that all updates that were made in the application in the user's session are saved on disk.
- Unlike logout, the application stops running. The next user that wants to use the application will need to restart it.

Errors

- In the application all errors and exceptions should be handled gracefully within the GUI setup. The text console should NOT be used at all: not to report any error, not to read input, not to print output.

Model

The model should include all data objects, plus code to store and retrieve photos for a user. The collection of classes that comprise the model should be in its own package, separate from the view and controller.

You are required to use the `java.io.Serializable` interface, and the `java.io.ObjectOutputStream`/`java.io.ObjectInputStream` classes to store and retrieve data.

See Notes on Serialization and Versioning to know how to implement serialization and deserialization.

Note that your application will need to store content for multiple users, so it would be a good idea to separate different user's contents from each other.

You need to think about what objects you want to have in your design, with what attributes and operations. It is important to plan this out and come up with a good object-oriented design that clearly separates roles and functions between objects, and can be cleanly extended to add more features for future versions of the application.

GUI Storyboard

Your first task is to design the UI in the form of a **storyboard**.

The storyboard is a sequence of screen diagrams that shows all paths of flow through the interface. Here's a sample storyboard for a calculator application that gives you an idea of what you should do. This is an older version built with Swing, so ignore the labels of the Swing widgets. Also, this is not a complete storyboard in that it does not show all possible screens that are in the UI it describes, nor does it necessarily show all possible transitions between screens. What it does show is how to draw screens, how to label screen components, how to draw transitions between screens, and how to label transitions.

It is important to have one or more overview diagrams that show all screens and all transitions between them, without any details of the components within the screens themselves. This is an overview that can give the complete picture in one shot. The rest of the storyboard will then draw each screen in detail.

Each screen must be drawn using some drawing package. Or, you can even include screenshots off SceneBuilder rendering of FXML UI layouts. But Screenshots that you take off a Java program WILL NOT be accepted. In other words, there should be **no Java code** written at this stage at all.

Each screen will represent one window of your GUI and will contain all the widgets that go into that screen - text fields, buttons, etc. Be as precise as you can about the selection and layout of the components in a screen. While it is not necessary that you label each Java FX component you will use (as in the sample storyboard), it will help you if you do because there is a smaller amount of design issues to worry about when you start coding the application logic.

Each screen will show transitions to other screens and the events that trigger these transitions. When all is said and done, you will have effectively drawn up a storyboard of your entire GUI that shows all screens and all inter-

screen transitions.

Note: The title for each screen should be descriptive of what the screen does. The sample storyboard says "Calculator" for all titles, but for your storyboard, name every screen with an appropriate title. This can serve as the title to be displayed in the titlebar when you implement these screens in code.

Grade Credit: Credit for the storyboard will be based on how well it anticipates (and determines) the implementation of the GUI. This portion of the grade for your storyboard will be given **after** you finish the implementation, and we can look at how useful your storyboard has been for your implementation. This means your storyboard is not set in stone, as in you can make some changes to the UI when you build it because you thought of some new/different elements. But the final result can't be too different from the original storyboard, which if it does, will imply that you didn't think through the UI well enough to start with.

Complete Implementation

Code your application using the standard installation of Java, using for your GUI Java FX and FXML **only** (No Swing). No external vendor libraries. We will test with standard Java so if you use any external packages, your program will not run, and you will not get credit.

Document every class you implement with Javadoc tags, and be sure to include authorship.

Represent the object-oriented design of the entire application using a UML class diagram. This should include both the classes you have built, as well as the Java FX classes. For the latter, just the class name will suffice--shade the class box so it's easy to visually separate the Java FX classes from yours. For each class built by your application, show all public fields and methods in the UML representation.

Keep in mind that you will need classes that are not visually represented, but perform data-management functions, as well as broker between the visual classes and the backend. These should be in your UML as well.

Finally, hand-drawn UML diagrams are NOT acceptable. You should use drawing software to do the UML. (Google Slides is an easy option, but if you find a UML drawing app, feel free..) The end product should be a PDF file.

Bitbucket Contents

Your project should be named PhotosXX, where XX is your 2-digit group number.

The **docs** and **data** directories mentioned below should be created directly under the project, NOT under **src** or under any of the packages.

By Sunday, Oct 25, 11 PM

GUI Storyboard: The final form of your storyboard should be a PDF file called **storyboard.pdf**, which should be placed in the **docs** directory.

By Wednesday, Nov18, 11 PM

- **Complete code**: There should be one class called **Photos** that should have the **main** method, so it can be launched as an application.
- **UML**: The complete UML class diagram should be a PDF file called **uml.pdf**, placed in the **docs** directory.
- **Javadocs**: The complete Javadoc HTML documentation should be generated and placed in the **docs** directory.
- **Stock Photos**: For the **stock** username. These should be in the **data** directory.

Grading

Your project will be graded on the following, for 225 points:

Category	Points
UI Design - Storyboard (Completeness)	20
Object Design - UML (Separation of functionality, proper relationships between objects, extensibility)	15
Features (including robustness)	190
Total	225

Penalties (up to 25 points total) will be assessed on the following:

- If you did not commit storyboard by Oct 25.
- Not using FXML adequately/appropriately to design the UI
- Inadequate Javadoc tags/Javadoc HTML documentation not generated
- Project structure does not properly separate model, view, control classes with appropriate package configuration
- Needing extra configuration on our part to test your project because you did not follow specifications
- Usability is poor such as roundabout ways to get at data, and/or confusing interface
- Lacks scalability i.e. doesn't display large amounts of data (e.g. many tens of photos or more) in a easily navigable way

Lateness penalties (separate from penalties above):

- 10 pts: Every time you ask us and we test another commit version in your repository that is earlier than the last commit before the deadline.
- 10 pts: For every 2 hours of lateness, in case there is nothing in the repository for us to test as of the deadline of Nov 18, 11 PM.
  NOTE: This 2 hour block will be applied STRICTLY starting any time after 11 PM (even if it is one second), in increments of 2 hours. NO EXCEPTIONS.