CS 4743
Applied Software Engineering
Spring 2017

Assignment 3: Audit Trail
Due: Monday, Feb. 20th 11:59pm

BACKGROUND:
By now, your assignment is probably starting to smell a bit ripe, at least with respect to loading views and accessing globally available objects. Before we add any more complexity to our program, we will refactor our project to centralize view loading. The result will be a much cleaner application structure that scales better (with respect to increasing functionality) and is easier to maintain.
Next: a common feature of enterprise applications is to use **audit trails** to track the changes made to important data objects, much like a log. This feature makes it easier for users to investigate the series of changes that have affected a given data object's current state.
So we will implement an audit trail for our author data object, including creating a database table, a model, a table view, and adding a new method to the author gateway. We will also convert the author gateway SQL to use transactions and include insertions into the audit trail. Lastly, we will introduce a modal dialog box to confirm deletion and prompt to save if we try to leave the author detail after changes have been made, and when deleting an author.

TASKS:
1. Implement a Singleton class to provide access to global data and behavior. You can create a new Class or modify an existing one (like the Application subclass).

A general object-oriented design rule to follow is if an object has state specific to a current run of the program, then that object should be instantiated (even if it is a globally-accessible object). This means that our program **should not use** public static member variables as a means for global access, but should instead provide the access through a globally-accessible object. And that object should be instantiated once, and only once. It can then make its member variables available to other classes using normal public accessors. This is the Singleton pattern.

We will go over the Singleton pattern in class and how to refactor your program's static member variables into a Singleton class. By making this change, you will also need to change all other classes that have previously referenced those public static member variables. Please note that if you use the Application subclass as a Singleton, it will not completely follow the classic Singleton pattern (e.g., the constructor must remain public).

2. Change Application member variables to a private member variables. You don't need accessors because only the Application will access these member variables (see the

next step).

3. Refactor the project to use a **changeView** method in the Application object. The goal of this change is a pretty important one. You have recognized by now that changing the content view of the stage involves several statements, involving loaders, controllers, models, and/or gateways. Some of these statements are the same for all views and some are different. But the overall logic is currently duplicated EVERY TIME we want to load a new view in the Center part of the stage's BorderPane. Additionally, the logic is now duplicated over multiple classes (MenuController and AuthorListController). What if later we need to change the fundamental logic for changing the content view? It is a plague of duplicated logic that will only spread and get worse.

There are lots of ways we can implement changeView, but I want you to do it my way so that you get some experience working with an Enum class. Enums are simple, efficient, and self-documenting. Here is the method signature I want you to use for changeView:

```
public boolean changeView(ViewType vType, Object data)
```

vType is a value from an Enum called ViewType. E.g., ViewType.AUTHOR_LIST
data is any model data that needs to be given to the controller. E.g., a list of Author models, or a single Author model for a detail view

Any object that needs to call changeView can get to it by calling the Application object's getInstance method. For example, in my car program, to display the detail for a blank Car model, my MenuController makes this call:

```
AppMain.getInstance().changeView(ViewType.CARDETAIL, new Car());
```

Once you finish this task, you should be able to sit back and admire a much more elegant, simple, and professional structure to your code. It is now less stinky.

4. Create the audit_trail table in your MySQL database. This table will provide a general purpose audit trail for author and book record changes. The table that an audit_trail entry applies to is indicated by its record_type value: "A" for author and "B" for book. The record_id is the foreign key into either the author or book table, based on the value of that audit trail entry's record_type. Your fields should be:

- id : int , primary key, auto-incrementing
- record_type : char(1), default 'A'
- record_id : int
- date_added : timestamp, default CURRENT_TIMESTAMP
- entry_msg : varchar(255)

The two defaults in your schema will automatically set the record_type to 'A' for an author audit trail entry and set the date_added to the current server date and time whenever you insert a new audit_trail record. The defaults can be overridden if desired.

Also, create an INDEX on the two fields record_type, record_id. Most/all of our queries against the audit trail table will use both of those fields in a WHERE or JOIN clause.

5. Create an Audit Trail Entry model that has 3 member variables:

- recordDescriptor : String
- dateAdded : Date
- message : String

recordDescriptor will store the full name of the associated author record (i.e., the concatenation of the first name, a space, and the last name).
dateAdded will store the value of the date_added table column of the record.
message will store the value of the entry_msg table column of the record.

Include appropriate constructors and accessors.

6. Implement the Audit Trail View and Audit Trail Controller. The Audit Trail View should have a Label or Text control that contains the text:
"Audit Trail for <author full name>", where <author full name> is the concatenation of the respective author's first and last name (note that you can use the value of the recordDescriptor member variable from the first AuditTrail model for this).

Under the Label or Text control should be a ListView that contains the following information per audit trail record:

- Timestamp of Message in "yyyy-mm-dd hh:mm:ss" format
- Message

The controller will populate the ListView with a list of Audit Trail Entry models passed to it in its constructor (similar to your Author List View).

7. Add a button on the author detail that will call your changeView method to load the Audit Trail View and an Audit Trail Controller. For new authors, no audit trail has been created yet. Therefore, it is reasonable for the program to display a modal dialog box with a message informing the user to save the author record first before clicking on the Audit Trail button. The JavaFX **Alert** class is a good way to display modal dialog boxes.

8. Add a method to the author table gateway that will fetch an author's audit trail (it can be called from the author model) and return a list of AuditTrailEntry models for the given author. Note that the select query responsible for the resultset will be a type of **JOIN** query since we need to pull data from 2 tables: author and audit trail. You may want to practice the query syntax in phpmyadmin before plugging it into Java, to make sure you have the syntax correct (easier to debug this way). Be sure to sort the resultset on date_added ASC from the query by adding an ORDER BY clause at the end of the

SELECT query.

Add a method to the Author model called getAuditTrail that will call the above gateway method. The Author model method will return the list of Audit Trail Entry models that are returned by the gateway.

9. Convert the author table gateway insert and update methods to use transactions and include insertion of an audit trail record stating the nature of the change. We should use a transaction to accomplish this because an audit trail record is inherently dependent upon the success of the preceding insert or update (i.e., they should be considered together as a single logical SQL statement). If the insert or update fails, then the audit trail insertion should not take place, and vice versa.

When inserting a new author, the audit trail message should be “Added”. When updating, add an audit trail record for every field whose value has changed. Note that you will have to first fetch a copy of the original record from the author table BEFORE it is updated. This is a great opportunity to add a new method to the gateway called getAuthorById that returns an instance of Author. You can call this method to get return an instance of the old author record data before you update it.

Examples:
If we are adding a new author with the name “Bob Smith” and a DOB of “1950-01-01”, there will only be 1 audit trail record that says: “Added”

Later, if correct Bob Smith’s last name to “Jones” and his DOB to “1950-05-10”, there will be 2 more audit trail records:
“Last Name changed from Smith to Jones”
“DOB changed from 1950-01-01 to 1950-05-10”

Also, be sure to use a transaction when deleting an author and delete audit trail entries for the author that is being deleted. Don’t let orphaned audit trail entries remain in the audit trail table.

10. Add a modal dialog call to confirm deletion (allows user to abort the deletion). Again, you can use JavaFX’s **Alert** class as a modal dialog box and determine which button was pressed when the Alert closes.

Also, change the delete method in your Author Table Gateway to first delete the author’s audit trail records AND then delete the author record. Put both of these statements in a single transaction.

11. Before a user navigates away from the Author Detail View, it is a good feature for our program to prompt the user to save his/her changes to that record IF the record has changed. The prompt should give the user the following choices:

a. Yes: save the record first, and then navigate away
b. No: don't save the record and then navigate away
c. Cancel: abort the view change and stay on the current view

Some simple rules for when this prompt should display are:
a. if the record is new (i.e., not yet inserted), always prompt the user to save if trying to navigate away from the detail view
b. if the record already exists but the values have not changed, then no prompt should be displayed and user can navigate away
c. if the record already exists but the values have changed, then prompt the user to save before allowing navigation to occr

There are a couple of techniques you can use to determine if a model's values have changed. You can compare the contents of the GUI fields to the model's values. If they are different then the user has changed something in the GUI. You can also add event listeners to the various GUI form fields and set a changed field to true when those event listeners fire (but you will have to set changed to false after saving).

The last design challenge is to determine where to put the logic for checking if the currently displayed view has changed, displaying a dialog box, and telling the current view to save BEFORE the view changes. But because you centralized the logic for changing views (right???), this should now be a straightforward, clean, and scalable enhancement to your program.

12. Lastly, any time the Author Detail View save method encounters an exception (e.g., a validator fails), display the message in a modal dialog box, informing the user of the problem.

DELIVERABLES:
1. Make this assignment its own Eclipse Java project called Assignment3. In your main Java class, include a comment with "CS 4743 Assignment 3 by <your name>".
2. Export the entire project to a zip file called assignment3.zip and submit the resulting zip file to Blackboard.

LATE POLICY:

-10 pts	few hours to 24 hours late
-20 pts	24 hours to 48 hours late
half credit	> 48 hours late

RUBRIC

20 pts	Refactor your Application object to be a Singleton
20 pts	Audit trail view, controller, and model implemented
15 pts	Author table gateway uses transactions for insert, update, and delete

10 pts Delete author displays a confirmation modal dialog allowing user to cancel the deletion
10 pts Author detail view prompts user to save when adding a new Author before navigating to different view
15 pts Author detail view prompts user to save when existing Author changes before navigating to different view
10 pts Author detail validation errors are now displayed in a modal dialog after clicking Save