Refactoring of DBReader

Contents

Purpose

In the previous version of DBReader we created a JavaFX application capable of viewing a specific table on MySQL (JDBC placeholder). This application had many limitations, like:

1. It can only display hardcoded tables properly.
2. Application is not easily upgradable as all the code is dumped in one class.
3. We need the user to know how to connect to a different Databases using the connection URL.

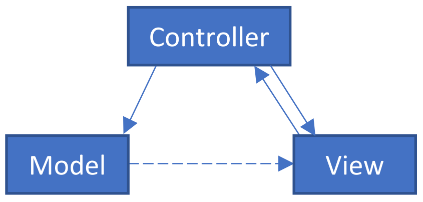
In this version, we will mitigate these issues.

Refactoring

Using some DPs (Design Pattern), we will attempt to refactor the code based on functionality. DPs allow the code to be more organized which in turn will make it easier to upgrade and maintain applications. Sometimes using DPs will require more coding and will make the code more complicated. However, the benefits of decoupling (Layering) and code organization forced by DPs greatly improve the upgradability and maintainability of applications.

We will use two DPs called MVC (Model View Controller) and Builder. MVC will handle the separation of JavaFX (GUI/View), JDBC (Model), and Logic (Controller). Builder will allow easier implementation of Connection URL to multiple DBs (only MySQL for this assignment).

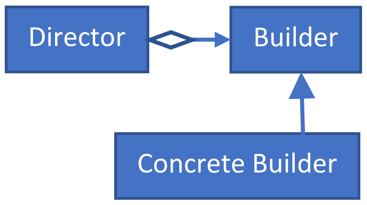
MVC



Model View Controller decouples the application in 3 components. Allowing each component to be theoretically independent of other components. There are variants of MVC which can have more than one of each component in different combinations.

- Model: Can be one class or a group of classes. It represents the underlying data and/or raw functionality. As an example, in our Assignment 1, MySQL is the model in a structural view. Model can update view. This is usually done not in a direct manner, rather in an indirect way usually though the Observer Pattern (to be mentioned later) or through the controller.
- View: Represents the visual representation of the Application. for example, JavaFX, Swing, or a Browser.
- Controller: Its jobs is to connect the view to model. Controller can update both view and model. There are different variations of how controller updates/manipulates other components. As an example, in Assignment 1 the whole server (including our code) is the controller.

Builder



This DP is used to "separate the construction of complex object from its representation so that the same construction process can create different representations"[1]. Director uses the Builder subclasses to build different objects under builder hierarchy.

The idea here is that the builder will be an interface or an abstract class with one or more concrete classes inheriting from it. These concrete classes are the full implantations of the Builder. For example, in this application the builder is the general JDBC builder and the concrete classes can be MySQL Builder (MySQL is the only one to be implemented), Oracle Builder, and Mongo Builder. These concrete classes will have specific instruction as how each URL connection must be built in respect to the Database.

The Director in this case will be JDBC Controller in our lab. The user will provide an instance of MySQLBuilder to the controller, MySQLBuilder is subclass of JDBCBuilder. Then the controller will ask builder to generate a URL based on the information it possesses or it was given to by user.

The main advantage will be the fact the director relays on Builder. Meaning we can pass different concrete builders to it and it will work the same way. Allowing for a greater flexibility in the application.

Observer

This DP is a notification system. It is used when multiple objects (Observers) are interested in changes of one Object (Observable). For example, a simple button is observable while all other objects which we add to button as action listeners are observers. Observers get notified of any changes in the observable object.

Observable is usually a class which can accept Observer interface as an argument. Whenever the state of the Observable changes, Observable will notify all Observers of the change. Assume Observer interface has one method called update and Observable class has a list of observers. Each time state of Observable object changes (state depends on implementation), Observable will notify all observers by looping through all observers and calling the update method. The argument/s of update method depends on the implementation.

In JavaFX we use the interface Property to handle the implementation of observer DP for us. For example, look at the example provided the assignment called PropertyExample.java. Every Node in JavaFX has many properties. These properties are width, height, text, color, and many others. Generally to access a property you will use a method named like [property name]Property():Property. Property methods start with the name of the property and end with the word Property. Almost all features in a Node (nodes are button, TextField, TextArea, etc.) have proprieties.

[1] (Design Patterns Elements of Resuable Object-Oriented Software, 1994, p. 97)

Requirements

Read the comments.

1. In ConnectDialog.java finish the constructor and methods init() and showAndWait().
2. In DBReader.java finish the methods init(), stop(), helper methods, and createOptionsBar().
3. Finish the JDBCController.
4. Finish JDBCURLBuilder and MySQLURLBuilder.
5. Finish the JDBCModel.
6. Create a sequence diagram of JDBCModel::search and JDBCModel::extractRowsFromResultSet.

Bonus

Difficulty increases as numbers go up as well as the bonus mark value. As usual attempt the bonuses when you are done with the requirements. It is up to you to figure them out.

1. Create a factory to get access to concrete instances of JDBCURLBuilder instead of calling new. The type of DB must be extracted from ConnectDialog.
2. Allow the cells in the table to be copied.
3. Create Junit for Model and/or Controller.
4. Complete the functionality of URL connection properties in the ConnectDialog. Currently the GUI components are there but not connected to controller. You will have to modify some of the base classes.
5. Allow the cells to be edited and update the DB. This is a major change to the project. I recommend create a class called row with properties bounded to each cell in the table. Using a list in the controller keep track of these updated rows, can be achieved using listeners. Finally add the update button which then forces the controller to go through each element and update them using the model.

Schedule

If you have time you should work on more contents in your early weeks. The less work you have left for last week of the term the better.

Week 1) Read the instruction, questions will be quiz based on these instructions.
Week 2) Start on View.
Week 3) Start on Model.
Week 4) Start on Controller and Builder.
Week 5) Do final cleaning and make your diagrams.

Submission

You will need to submit 1 zip file containing the files below:

1. Submit one **zip** file of your project.
   [firstName]-[lastName]-[labSection#].zip
   ex: shawn-emami-11.zip
2. Submit one **jar** file of your project.
   [firstName]-[lastName]-[labSection#].jar
   ex: shawn-emami-11.jar
3. Submit your diagram/s.

There is no demo for this assignment. Just submit to Bright Space.

Export Instruction Slides

This is a maven project, to export build your project with maven. There will be a zip and a jar file inside of the target folder of your project.

Instructions

Approach

These instructions will explain how to refactor your code. Below is the highlight of each step. There is no specific order in these steps. However, each step has suggestions as how it should be approached.

1. View, this will consist of 2 classes mostly done. Some work will be needed to complete it. This is an example of refactored lab 1. This should be done first so while implementing other steps you can constantly run your application and test it.
2. Controller, this class is dependent on Model. However, it can be created in advance with empty methods. Controller will delegate complex works to model. It deals with the builder and act as abstraction between view and model.
3. Builder, this component is stand alone and can be worked on as first or last step. It can also work and be tested interpedently. The main purpose here to provide a way to create URL connection based on given information.
4. Model, this component is stand alone and can be worked on as first or last step. It can also work and be tested interpedently. The main purpose is to keep all the code related to DB communication. There is no specific logic, methods have different functionalities.

View

Majority of the JavaFX code is already done. Your job is to clean up some leftover methods and implement the Controller into the view.

If in your view you get access restricted under your javafx import set the JRE of your project to workspace. To do this, in you project tree right click the “JRE System Library” and choose properties. Then Select the third bulletin and click “Apply and Close”.

To find all the //TODO tags go to Window/Show View/Tasks. If you cannot find Tasks then instead click on Window/Show View/Other, in the search box type Tasks.

ConnectDialog

You are to finish the constructor and the methods init and showAndWait. Make sure to read the //TODO tags in the code for more instructions.

DBReader

You are to finish the methods init(), stop(), and createOptionsBar(). Make sure to read the //TODO tags in the code for more instructions.

Constroller

This class is very simple for the majority you are just calling the equivalent method on model. Look at the class diagram for details of all methods.

1. Methods close, search, getAll, getColumnNames, getColumnNames, and isConnected are straight forward delegation.
2. Method getTableNames, if model is connected clear tableNames and then call addAll on it and pass to it the result of model::getTableNames.
3. Methods setDataBase, addConnectionURLProperty, and setCredentials call equivalent methods on builder.
4. Connect method will get the url from builder and pass it to conntcTo of model.
5. tableInUseProperty is a getter.
6. In constructor, initialize the tableNames with FXCollections. Initialize model and tableInUse. Make sure to look at the PropertyExample.java. Add a listener to tableInUse and inside of it call getAndInitializeColumnNames on model. Pay attention to what argument you passing to it.

Builder

This DP and its classes will be placed inside of the **jdbc.builder** package. This builder will be used to create the connection URL needed to connect to MySQL db. For example:

jdbc:mysql://localhost:3306/redditreader?serverTimezone=UTC

The connection URL above is composed of 6 components, JDBC, DataBase, Address, Port, Catalog, and everything after the "?" which are the properties. The job of builder is to create the final string like above based on given components.

Builder will be created by the user and given to JDBCModel. All other data such as address, and port will also be given to model. Model then uses the builder by passing all the needed information to it and generate the connection URL.

1. Create a new package called **jdbc.builder**.
2. Review the UML Diagram Attached for the builder package.
3. Inside of this package you will have to create two classes, **JDBCURLBuilder** and **MySQLURLBuilder**. Look at your class diagrams for more details for each class.

JDBCURLBuilder Class

Use the Class Diagram to complete the given class. All methods are basic setters, meaning simply set the variables. Pay close attention to accessors, final (bold), abstract (italic), static (underlined), and names. In the constructor make sure to initialize your map with the HashMap class.

Error Checking

The only error checking you will need to do is null check and port cannot be negative. Have in mind properties are optional, meaning they might not be added to the builder. However, values being added still cannot be null. You can use one the methods below for null checking:

1. If [variable] is negative throw an illegal argument exception.
2. If [variable] is null throw a null pointer exception.
   a. You can also use the utility method **Objects.requireNonNull([variable], [exception message])**.

MySQLURLBuilder Class

Use the Class Diagram to complete the given class. In the constructor make sure to call **setDB(String):void** method and pass to it **"mysql"**. This is to initialize the type of DB we will be using. Next you need to override the abstract method **getUrl():String**. In this method you will want to assemble (concatenate) a connection URL like the example below:

<u>jdbc</u>**:**<u>mysql</u>**://**<u>localhost</u>**:**<u>3306</u>**/**<u>redditreader</u>**?**<u>serverTimezone=UTC</u>**&**<u>useUnicode=true</u>

Each of the components in this URL are stored in the instance variables of the super class JDBCBuilder. For example:

1. **Jdbc** is stored in the static final variable **JDBC**.
2. **mysql** is stored in variable **db**.
3. **localhost** is stored in variable **address**.
4. **3306** is stored in variable **port**.
5. **redditreader** is stored in variable **cataglog**.
6. Any thing after "**?**" is stored in variable properties.
    - properties is a map. It stores data in pairs like a dictionary. In a dictionary each word (key) is associated to a definition (value). A map stores information based on keys and values. For example if I want to put something inside of a map I can use **properties.put([key], [value]);**. Map is an interface in java and HashMap is the concrete class.
    - In this example "**serverTimezone**" will be the key and "**UTC**" will be the value.
    - There can be multiple properties and they are separated using the "**&**" symbol in the URL.

Inside of the getUrl method follow the following steps. Make sure to use a StringBuilder, if you want you can use String::format with StringBuilder::append for step 2 instead of using append 8 times.

1. Create a StringBuilder object.
2. Using the StringBuilder::append method, append the variables (1-5) described above with :, ://, :, and /.
3. If properties variable is not empty (use isEmpty()) then using append method, append step 6 from above using & for between properties.
4. Finally return builder.toString()

Model

Look at the class diagram and complete the signature of all methods first. This class is focused on providing functionality of manipulating the DB. Some methods are also explained in sequence diagram.

1. In constructor initialize the lists.
2. setCredentials Is just a setter.
3. checkConnectionIsValid, throw an exception if connection is null or closed.
4. connectTo, if the connection is open close it first and then use DriveManager to get a new connection. Use the existing method.
5. getAll, is just search with no search term.
6. close, if connection is not null close the connection.
7. buildSQLSearchQuery, you want to assemble a SQL Query. When doing the following steps be mindful of spaces between words as I might not have mentioned all of them.
    a. Make a String builder and initialize with “select * from ”, since this will be common for all searches.
    b. Append the table name.
    c. If there are no parameters return the String in StringBuilder.
    d. Append “where”.
    e. Now for each columnName append column name, “like ?”, and “or”.
        i. We want to search all columns. The final result for example will look like: “select * from account where id like ? or user like ? or pass like ?”
    f. Finally return the string in StringBuilder.
8. extractRowsFromResultSet, this method executes the query of PreparedStatement in a try and catch. Then while result set has next, create a new list called row and add to it the content of each row using ResultSet::getObject. Finally add the row to the list passed as argument. You can use the code in solution of the lab 1.
9. search, in this method PreparedStatement is created and executed.
    a. Check the connection is valid.
    b. Check table and column names are valid.
    c. Create a two dimensional java.util.List of type Object, initialized with LinkedList.
    d. Call buildSQLSearchQuery and store the results.
    e. In a try with resource create PreparedStatement from connection. You can use the code from lab 1 solution.
    f. If search term is not null
        i. searchTerm = String.format("%%%s%%", searchTerm);
           this will produce the search term with one % before and after. Extra ones are for escaping.
        ii. For the size of columnNames call setObject on PreparedStatement and pass index and searchTerm to it.
    g. Call 8. extractRowsFromResultSet.
    h. Return the two dimensional list.