CPSC 2151

Lab 10

Due: Friday, April 2nd at 10:00 pm

In this lab, you will be working with the Model-View-Controller architectural pattern. While the Model-View-Controller pattern will be beneficial for working with Graphical User Interfaces (GUI), we will not be working with a GUI for this assignment yet. We will be building off the code from a previous lab, specifically the `Mortgage` and `Customer` interfaces and classes.

Instructions

You will need to create a new project with a package called `cpsc2150.banking`. Add your `Mortgage` and `Customer` classes and interfaces from lab 6 to that package. (**Note:** You will need to create the `models` folder for these files to match their package structure.) You will then need three more classes:

MortgageApp.java

`MortgageApp` will be a very simple class. It will contain our `main` method that will be the entry point of the program. All it will do is declare an instance of `IMortgageView` and an instance of `IMortgageController`, and then call `MortageController.submitApplication()`. The code should look like this:

```
package cpsc2150.banking;

public class MortgageApp {

    public static void main(String [] args) {
        IMortgageView view = new MortgageView();
        IMortgageController controller = new MortgageController(view);
        view.setController(controller);
        controller.submitApplication();
    }
}
```

This type of setup is common when using MVC architectural pattern in Java. We just need the entry point to set up our **Controller** and **View**, then turn over control of the program to the **Controller**.

MortgageView.java

`MortgageView` will serve as the **View** layer of our MVC architecture. That means that `MortgageView` is the only class that can get input from the user or print to the screen. The **Controller** layer will use `MortgageView` to perform these tasks. `MortgageView` also does not know about our **Model** layer, so it cannot perform any input validation on whether or not the given values meet our `Mortgage` or `Customer` classes' preconditions. `MortgageView` can still validate whether a "Y" or "N" was entered for yes or no prompts since that is a requirement of the **View** itself, not the **Model**.

The only class-level variables that `MortgageView` should have are a `Scanner` object and its `IMortgageController` object. Remember, we can have issues if we have multiple `Scanner` objects looking at the same input source, so we don't want to declare one in each function. We actually won't need to use the `IMortgageController` object, but the initialization ensures clause says we must include it

`MortgageView` should implement the `IMortgageView` interface using a command-line prompt for the user interface. The "get" methods all ask the user for the specified value. They do not perform any data validation (except for validating the Y/N response) as the view does not have access to the **Model** layer. We can assume the user will input the correct data type (i.e., a number when a number is expected).

Note 1: You may not need to use all the functionality that `IMortgageView` provides to complete this assignment. That is completely fine; the interface was written to be used for multiple programs, so it may include extra functionality that we don't need right now.

Note 2: The `IMortgageView` and `MortgageView` files should be placed inside a new `views` folder to match the package `cpsc2150.banking.views`.

MortgageController.java

The `MortgageController` class implements `IMortgageController` and serves as the **Controller** layer in our MVC architecture. It controls the flow of control in our program. This class can use both the **View** and the **Model** layers to accomplish this task. This class will check to make sure the data that the user provides (through the **View** layer) meets the preconditions that exist in our **Model** layer. If the input is not valid, the **Controller** will (through the **View**) alert the user to the error and re-prompt the user for correct input. The `MortgageController` class will also use the **Model** layer (`Mortgage` and `Customer` interfaces and classes) to actually apply for the mortgage. After a mortgage has been applied for, the **Controller** will (through the **View** layer) display the `Customer` and `Mortgage` information to the user (**Note:** use the `toString` methods to do this).

The controller will allow for a customer to apply for multiple loans (without having to re-enter the customer information) and allow for the user to enter information about multiple customers. Remember, the **Controller** layer cannot directly interact with the user, it must go through the **View** layer.

The `MortgageController` class will have one `private` field, an `IMortgageView` object, which is set by an argument passed into the constructor. The `MortgageController` class will have one `public` method, called `submitApplication()` which runs the program.

All the logic concerning validating the **Model's** preconditions, and the order of events, is handled by the **Controller** layer. The logic concerning the `Mortgage` application or the `Customer` itself are handled by the **Model** layer.

Note: The `IMortgageController` and `MortgageController` files should be placed inside a new `controllers` folder to match the package `cpsc2150.banking.controllers`.

TIPS and additional Requirements

- You must provide a `makefile` with `make`, `make run` and `make clean` targets.

    - This will be a large deduction if you do not
- You do not need to provide any automated testing, although you should test your program.
- You do not need to provide any contracts for `MortgageView`, `MortgageController`, or `MortgageApp`. `Mortgage` and `Customer` should have contracts from a previous assignment.
- You **DO** need to comment your code
- You need to follow the Model-View-Controller architectural pattern for this assignment.
    - All interaction with the user goes through the **View** layer. **View** does not have access to the **Model** layer.
    - **Controller** layer handles input validation, and the order of events of the program. The **Controller** layer has access to the **View** and the **Model**, so it is the go between for those two layers
    - The **Model** layer handles our entity objects and the "lower level" logic. It does not have any access to the other two layers.
- Follow our best practices: No magic numbers, information hiding, separation of concerns, etc.
- `Customer.java` has a `toString` method that will be useful.
- Make sure your package structure is correct. If IntelliJ / Java gives you an error, check to see if your constructors and functions have the correct visibility!

Partners

You are required to work with one partner on this lab assignment. Make sure you include both partners' names on the submission. You only need to submit one copy. Remember that working with a partner means working *with* a partner, not dividing up the work.

Before Submitting

You should make sure all of your code will run on SoC Unix before you submit. Make sure you correctly set up the directory structure to match the package name. You should use the `makefile` provided in Lab 6 as a starting point, modify it so it works for Lab 10 and include it in your submission. **There is no need for you to include the JUnit test plan for this lab.** No late submissions will be accepted.

Submitting your file

You will submit your files using handin in the lab section you are enrolled in. If you are unfamiliar with handin, more information is available at https://handin.cs.clemson.edu/help/students/. You should submit a zipped directory with your package directory and your `makefile`. The TA should be able to unzip your directory and use any of the targets as specified above.

NOTE: Make sure you zipped up your files correctly and didn't forget something! Always check your submissions on handin to ensure you uploaded the correct zip file.

Sample input and outputs:

```
What's your name?
Jim Halpert
How much is your yearly income?
```

```
-12000
Income must be greater than 0.
How much is your yearly income?
56000
How much are your monthly debt payments?
-247
Debt must be greater than or equal to 0.
How much are your monthly debt payments?
247
What is your credit score?
-800
Credit Score must be greater than 0 and less than 850
What is your credit score?
1000
Credit Score must be greater than 0 and less than 850
What is your credit score?
748
How much does the house cost?
-120000
Cost must be greater than 0.
How much does the house cost?
120000
How much is the down payment?
-10000
Down Payment must be greater than 0 and less than the cost of the
house.
How much is the down payment?
130000
Down Payment must be greater than 0 and less than the cost of the
house.
How much is the down payment?
13000
How many years?
-15
Years must be greater than 0.
How many years?
30
Name: Jim Halpert
Income: $56000.0
Credit Score: 748
Monthly Debt: $247.0
Mortgage info:
Principal Amount: $107000.0
Interest Rate: 4.5%
Term: 30 years
Monthly Payment: $542.1532815136978

Would you like to apply for another mortgage? Y/N
```

```
y
How much does the house cost?
125999
How much is the down payment?
12000
How many years?
15
Name: Jim Halpert
Income: $56000.0
Credit Score: 748
Monthly Debt: $247.0
Mortgage info:
Principal Amount: $113999.0
Interest Rate: 4.0%
Term: 15 years
Monthly Payment: $843.2368383152977

Would you like to apply for another mortgage? Y/N
n
Would you like to consider another customer? Y/N
y
What's your name?
Dwight Schrute
How much is your yearly income?
73000
How much are your monthly debt payments?
...
```