C258 - Full Stack Development Using Java

Assessment: Flooring Mastery

Overview

The goal of this mastery project is to create an application that can read and write flooring orders for TSG Corp. Your complete application must demonstrate your understanding of all the topics covered in the prior sections, including:

- N-tier/MVC architecture, including the use of a service layer
- Interfaces
- Spring dependency Injection
- Unit Testing

We have covered all of the technical skills needed to complete this assignment. It is your job to think critically about this project, ask clarifying questions where requirements and features are not clear, and come up with a solid design and game plan before you begi coding. It is essential to your success that you don't just "throw code" at the problem and hope that something sticks. We want you use the background skills and techniques covered so far to synthesize a solution to this problem.

Architectural Guidance

1. Use N-tier development and MVC principles in structuring code to handle products, taxes, and orders appropriately. Your application should follow the MVC pattern presented in the course.
2. Use unit tests and integration tests to ensure that your application's data layers and business logic (service layer) code is covered.

Rules of Flooring Mastery

1. We are using an enterprise MVC architecture for this project, so your code must be organized into reasonable classes. You will draw a UML class diagram and high-level flowchart before proceeding with writing code.
2. For an enterprise architecture, our code must have layers:
    - The `model` package may only contain classes that have data members (properties).
    - The `dao` package contains classes that are responsible for persisting data.
    - The `controller` package contains classes that orchestrate the program.
    - The `view` package contains classes that interact with the user.
    - The `service`package contains the service layer components.
    - The `UserIO` class (along with the view component) will handle all console IO for the user.
3. Build this application following the process outlined in the *Agile Approach Checklist for Console Applications*

Requirements

Sample Data Files: **SampleFileData.zip**

The application will use 3 separate file formats for information.

Orders Files & Format

All orders are stored within an Orders folder. A new order file is created for each sales day. The file name is guaranteed to be unique for that day because the date is part of the file name. When creating new order files the file name should be in this format: `Orders_MMDDYYYY.txt`.

With this format, order file for Augusst 21st, 2017 would be named `Orders_08212017.txt`.

The order file should contain a header row. Data rows will contain the following fields and should be in this order:

1. OrderNumber – Integer
2. CustomerName – String
3. State – String
4. TaxRate – BigDecimal
5. ProductType – String
6. Area – BigDecimal
7. CostPerSquareFoot – BigDecimal
8. LaborCostPerSquareFoot – BigDecimal
9. MaterialCost – BigDecimal
10. LaborCost – BigDecimal
11. Tax – BigDecimal
12. Total – BigDecimal

Here is a sample line from an orders file:

```
1,Ada Lovelace,CA,25.00,Tile,249.00,3.50,4.15,871.50,1033.35,476.21,2381.06
```

`Orders_06012013.txt` is a sample file which is included in the sample download above.

Some of the order fields are calculations:

- MaterialCost = (Area * CostPerSquareFoot)
- LaborCost = (Area * LaborCostPerSquareFoot)
- Tax = (MaterialCost + LaborCost) * (TaxRate/100)
    - Tax rates are stored as whole numbers
- Total = (MaterialCost + LaborCost + Tax)

Tax File & Format

The tax information can be found in `Data/Taxes.txt`. This file will contain the following fields:

1. StateAbbreviation – String
2. StateName – String
3. TaxRate – BigDecimal

Here is a sample line from the taxes file:

```
TX,Texas,4.45
```

Products File & Format

The current product information can be found in `Data/Products.txt`. It contains the following fields:

1. ProductType – String
2. CostPerSquareFoot – BigDecimal
3. LaborCostPerSquareFoot – BigDecimal

Here is a sample line from the products file:

```
Tile,3.50,4.15
```

User Stories

The UI should start with a menu to prompt the user for what they would like to do:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * <<Flooring Program>>
 * 1. Display Orders
 * 2. Add an Order
 * 3. Edit an Order
 * 4. Remove an Order
 * 5. Export All Data
 * 6. Quit
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Display Orders

Display orders will ask the user for a date and then display the orders for that date. If no orders exist for that date, it will display an error message and return the user to the main menu.

Add an Order

To add an order will query the user for each piece of order data necessary:

1. **Order Date** – Must be in the future
2. **Customer Name** – May not be blank, allowed to contain [a-z][0-9] as well as periods and comma characters. "Acme, Inc." is valid name.
3. **State** – Entered states must be checked against the tax file. If the state does not exist in the tax file we cannot sell there. If the tax file is modified to include the state, it should be allowed without changing the application code.
4. **Product Type** – Show a list of available products and pricing information to choose from. Again, if a product is added to th file it should show up in the application without a code change.
5. **Area** – The area must be a positive decimal. Minimum order size is 100 sq ft.

The remaining fields are calculated from the user entry and the tax/product information in the files. Show a summary of the order once the calculations are completed and prompt the user as to whether they want to place the order (Y/N). If yes, the data will be added to in-memory storage. If no, simply return to the main menu.

The system should generate an order number for the user based on the next available order # (so if there are two orders and the ma order number is 4, the next order number should be 5).

Edit an Order

Edit will ask the user for a date and order number. If the order exists for that date, it will ask the user for each piece of order data bu display the existing data. If the user enters something new, it will replace that data; if the user hits Enter without entering data, it will leave the existing data in place. For example:

```
Enter customer name (Ada Lovelace):
```

If the user enters a new name, the name will replace Ada Lovelace. If the user hits enter without entering any data, it will leave the data as-is.

Only certain data is allowed to be changed:

- CustomerName
- State
- ProductType
- Area

If the state, product type, or area are changed, the order will need to be recalculated. Order date may not be changed!

After querying for each editable field display a summary of the new order information and prompt for whether the edit should be saved. If yes, replace the data in the file then return to the main menu. If no, do not save and return to the main menu.

Remove an Order

For removing an order, the system should ask for the date and order number. If it exists, the system should display the order information and prompt the user if they are sure. If yes, it should be removed from the list.

Quit

On quit, exit the application.

Stretch Goal: Export Data

Selecting this option should save all active orders files to a file called *DataExport.txt* within a *Backup* folder. Exporting the order data multiple times should overwrite the data within the file with the latest active order information. However, unlike other order files, an order's line item in this DataExport file should also include the date in MM-DD-YYYY format, and the file's header should reflect this addition.

For example, the a line of an example export, showcasing an order on June 12th, 2013:

```
1,Ada Lovelace,CA,25.00,Tile,249.00,3.50,4.15,871.50,1033.35,476.21,2381.06,06-12-2013
```

Additional Guidance

- Any time a user enters invalid data, the system should ask them again until they enter valid data. A user should not be able to crash the application with invalid data.
- We are using a layered architecture for this project. Thus your code must be organized into reasonable well-named classes an split into correct packages.
- Please submit your project plan with your code. This can be flowcharts, outlines, pseudocode, or any other documentation an planning technique.
- You must unit test your DAO and ServiceLayers.

Some Tips

- Build this application following the process outlined in the *Agile Approach Checklist for Console Applications* (as specified in the Rules of Flooring Mastery section above). This process will help you approach the project in a logical and orderly manner.
- If you get stuck, ask an instructor for general guidance on what you should be looking for. The instructor will not write the cod for you, but advice is always free.
- There are places where this specification has been left open to interpretation, this is intentional. When on the job you will frequently have to ask clarifying questions to business stakeholders. Be sure to ask a staff member for general guidance on any features you do not understand.
- Try to work on pieces of functionality in isolation and test them before moving on. Working on too large of a chunk at once leads to difficulties in troubleshooting and poor separation of concerns.
- Think, then code. If you do not have a plan for each workflow, you will not be able to complete this project.
- Use prior code samples and assessments as inspiration for this project. At this point in the course you should be realizing tha there are patterns for creating a layered data-drive application.

Submitting Your Assessment

Submit your files using instructions provided by your instructor.

After submitting your work, schedule a time with a staff member to review your code. If you are attending this class in person, your code will be reviewed during the weekly code review.