OOP Related task description using MVC

|  |  |
| --- | --- |
| **Done by:** | **Waleed Ahmad** |

# **Task 1 – Analysis of Operation Classes**

## Classes Purpose

The "Operations" interface establishes a contract that all operations must adhere to, making sure they all employ the same methods. If the operations adhere to this contract, they can be used interchangeably.

An abstract class called "AbstractOperation" implements the shared variables and methods of the Operations interface, thereby reducing code duplication and making it simple to add new shared variables or methods.

Two classes that use the "Operations" interface are "DeleteOperation" and "AddOperation." They give specific implementations for the "execute()" method, which tells how the corresponding operation should be run. For example, the "execute()" method of "DeleteOperation" can remove an item from a list, while the "execute()" method of "AddOperation" can add an item to the same list. These classes provide the core functionality for operations like deleting and adding and make it easy to reuse code.

The Java class "JComponent [1]" provides starting point for building GUI elements like buttons, text fields, and labels. It is a compact container with methods for managing component positioning, appearance, and input event handling that derives from the Container class.

"DeleteUI" and "AddUI" are two classes that provide the user interface for removing and adding items, respectively. They are based on the "JComponent" class and have an instance of the operation class that goes with them (i.e., "DeleteOperation" or "AddOperation"). The goal of these classes is to let the user interact with the corresponding operation and set any necessary parameters. For example, the "DeleteUI" class might show the user a list of items and let them choose which one to delete. The "AddUI" class may give the user text fields or other ways to enter information about the item they want to add. These classes make it easy for the user to delete and add things, and they make sure that the user interface is the same for all kinds of operations.

"OperationFactory" class: The purpose of this class is to create instances of operations based on an input parameter, using the DeleteUI and AddUI classes to show the right user interface. This class acts as a hub for making new operations, making it easier to manage and keep track of a lot of them. By using the UI classes to show the right user interface, all operations can give the user the same experience. This class also cuts down on duplicate code by making it easier to add new operations from one place.

## How these classes fit together?

The "Operations" interface defines a set of methods that any operation class must implement. The "AbstractOperation" class provides a partial implementation of these methods to make it easier to make new operation classes. The "DeleteOperation" and "AddOperation" classes are concrete implementations of the interface. The "DeleteUI" and "AddUI" classes, which inherit from "JComponent," provide the user interface for deleting and adding items, respectively. The "OperationFactory" class makes instances of the UIs and the operations that go with them and links them together.

## Why classes extend from ancestors?

The "AbstractOperation" class adds to the "Operations" interface by implementing some of the interface's methods. By doing this, the amount of code that needs to be repeated in concrete implementations of the interface is cut down. This makes it easier to make new operation classes.

"DeleteOperation" and "AddOperation" both extend from "AbstractOperation" because they need to implement the "Operations" interface methods for deleting and adding data, respectively. By extending "AbstractOperation," they get a partial implementation of those methods.

"DeleteUI" extends from both "DeleteOperation" and "JComponent" to get their properties and functions. By adding to the "DeleteOperation" class, it can use the methods for deleting data. By extending "JComponent," it gets access to the base class for all Swing UI [2] components. This lets it inherit and change the standard UI component's properties and methods. Also, "DeleteUI" can be used as a UI component for deleting data because it is based on both classes.

"AddUI" extends both "AddOperation" and "JComponent," so it gets their properties and functions. By inheriting from "AddOperation," it can use the methods for adding data. By extending "JComponent," it gets access to the base class for all Swing UI components. This lets it inherit and change the standard UI component's properties and methods.

"OperationFactory" extends from both "DeleteUI" and "AddUI" to get their properties and functions. By extending "DeleteUI" and "AddUI," it gets access to their methods and UI components for deleting and adding data, respectively. Also, because "OperationFactory" extends both classes, it can create instances of both UI components and their operations and link them together. This makes it easy to make and use operations, user interfaces, and classes that are related to them.

## Advantages

* The architecture keeps the logic for operations and the logic for the user interface separate. This makes it easier to maintain and change each component on its own.
* The use of interfaces and abstract classes allow for flexibility in adding or modifying operations and UI components, as they can be easily extended or implemented.
* The architecture makes it easier to reuse code because different parts of the project or even other projects can use the same components.

## Disadvantages

* Using multiple classes and relationships can make the architecture hard to understand, especially for new developers who join the project.
* The use of interfaces and abstract classes can add extra work to the architecture, which can slow things down in some situations.
* : Developers may need more time to understand and learn the architecture, especially if they are not familiar with the design patterns used.

# **Task 2 – MVC Architecture Analysis**

Model View Controller architecture of application is as following:

**Model**: it contains the Model interface, which tells classes that want to be the application's model what methods they need to implement. The Model class uses the Model interface, which is in charge of keeping the application's state up to date. It has a group of Record objects and methods to add, remove, and change records in that group. Any changes to the records are sent to the view by the model.

**View**: it contains the View interface, which tells classes that want to be the application's view what methods they need to implement. The View interface is used by the MainWindow and OperationUI classes. The records are shown in a table format in the application's main window, which is made up of the MainWindow class. When a user wants to add or change a record, a dialog box with the OperationUI class is shown. When the model changes, the view makes changes to the user interface. The display(model) method of the MainWindow class changes the table when the model changes. The data from the record being edited are added to the data in the dialog box by the OperationUI class.

**Controller**: it has the Controller class, which is the middleman between the model and the view. The doOperation() method is used by the Controller class. This method is called when the user does something that changes the model. The Controller class makes a new Operation object that holds the model operation that needs to be done. The Operation object then makes an instance of the OperationUI class, which shows the settings needed for the operation. The user's settings are taken in by the OperationUI class, which then calls the Operation object's execute() method. The model is then changed by it.

## User Pressing Apply Button

When the user clicks the APPLY button on the ADD operation's dialog box, the following calls are made:

* doApplyButton() method in OperationUI is invoked, which first sets the dialog's visibility to hidden, and then runs the execute() method of the Operation that was supplied to the constructor of OperationUI.
* The execute() method of AddOperation class is invoked, retrieving the values that the user entered in the dialog fields, and using those values to create a new Record object.
* The newly created Record is added to the collection of records by invoking the add() method of the Model class and passing it as an argument.
* When the update() method of MainWindow class is invoked, the records panel is updated with the new record by invoking its display() method.

As a result, the user's new record is added to the collection of records in model , and the records panel is updated in view to reflect the change.

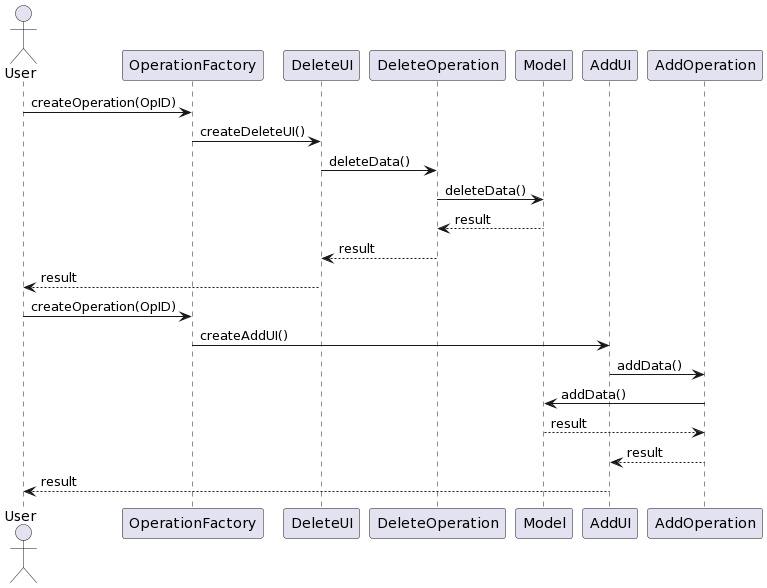


Figure : Sequence Diagram

## Advantages

* A distinct separation of concerns is made possible by the use of the MVC pattern, which makes the code simpler to read, comprehend, and maintain.
* It is simpler to test the code because the business logic is separated from the user interface.
* The code is more modular and reusable because of the use of interfaces and the separation of concerns.

## Disadvantages

* The MVC pattern's use increases the codebase's complexity, which can be too much for inexperienced developers to handle.
* The MVC pattern can be viewed as over-engineering in some situations, especially for smaller projects with simpler requirements.
* Due to increased communication overhead between the various parts of the code because of the separation of concerns, the code may become slower and less effective.

# **Task 3 – Additional Operations Implementation**

## Replenish Stock

The execute and settingsUI methods are overridden by the replenishStockOperation, which implements the AbstractOperation interface. An instance of the replenishStockUI, which asks the user for the product ID and quantity to add to the stock, is returned by the settingsUI method. With the supplied product ID and quantity, the execute method calls the replenishStock method of the Model. Code for replenishStockOperation is as following:

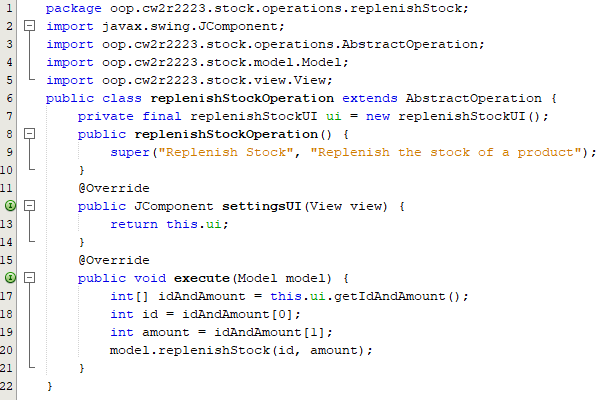


Figure : replenishStockOperation code

replenishStockUI is a JPanel [3] that has two text fields for the product ID and amount. The product ID and amount entered by the user are returned in an array of integers by the getIdAndAmount method, which is also available. Code of replenishStockUI is as follows:

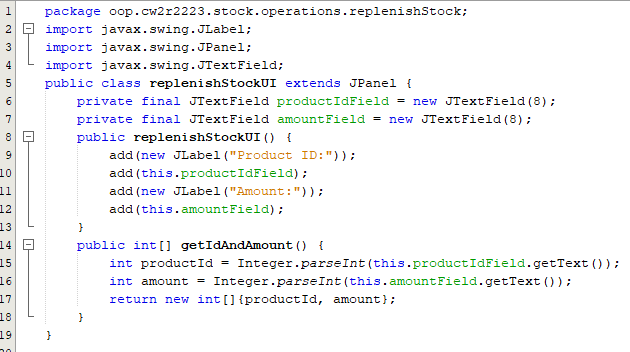


Figure : replenishStockUI code

## Apply Discount

The "SetDiscountOperation" class represents an operation to set a discount on all items in a specific department and is an implementation of the "AbstractOperation" abstract class. It has a "SetDiscountUI" object that prompts the user for the discount percentage and department name, then uses the "applyDiscount" method of the "Model" class to apply the discount to the items in that department.

A with the "SetDiscountUI" is a Swing “JPanel" class displays two input fields for the department name and discount percentage. It includes techniques for obtaining the values entered in these fields and converting them into the correct types. Code of SetDiscountOperation and SetDiscountUI are as follows:

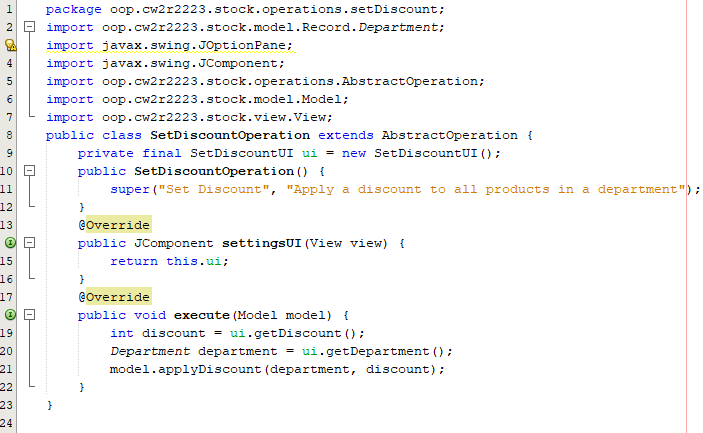


Figure : SetDiscountOperation code

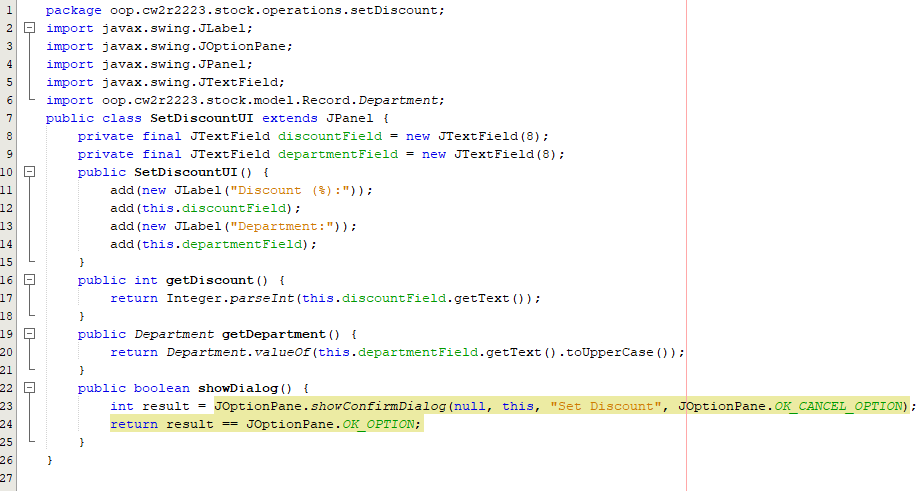


Figure : SetDiscountUI code

## Update Department

The user is prompted to enter a product ID and a department to set when using the "UpdateDepartmentOperation" class, which extends "AbstractOperation". A "JPanel" with text fields for the product ID and department is the result of the "settingsUI()" method. The "execute()" method calls the "updateDepartment()" method on the Model object with the product ID and department after retrieving them from the UI and converting the department name into a "Department" object.

The "UpdateDepartmentUI" class extends "JPanel" and offers a straightforward user interface with text fields for the department and product ID. The product ID and department are retrieved from the text fields by the "getIdAndDepartment()" method and returned as a string array. Code of both classes is as follows:

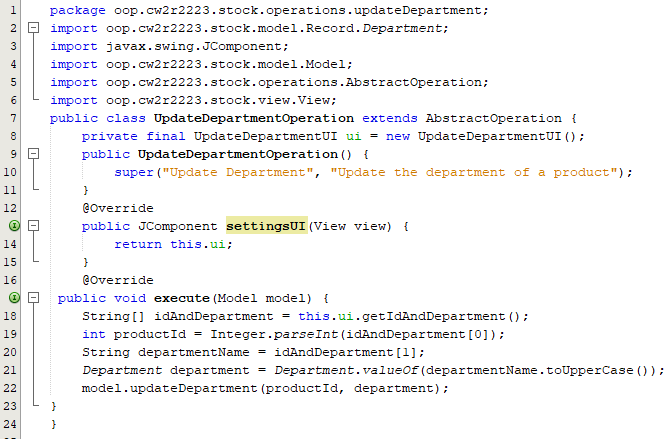


Figure : UpdateDepartmentOperation code

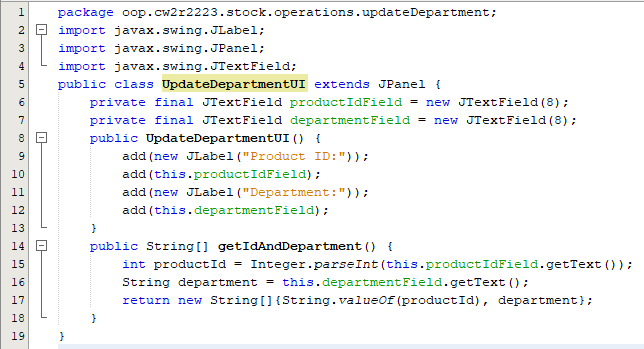


Figure : UpdateDepartmentUI code

## Set Discount

The user is prompted to choose a department and enter a discount percentage when using "SetDiscountOperation," which then calls the "applyDiscount()" method on the "Model" with the department and discount they have chosen. The user can enter the department and discount in a dialog box provided by "SetDiscountUI"; these details are obtained and used by "SetDiscountOperation". Code of both classes as following:

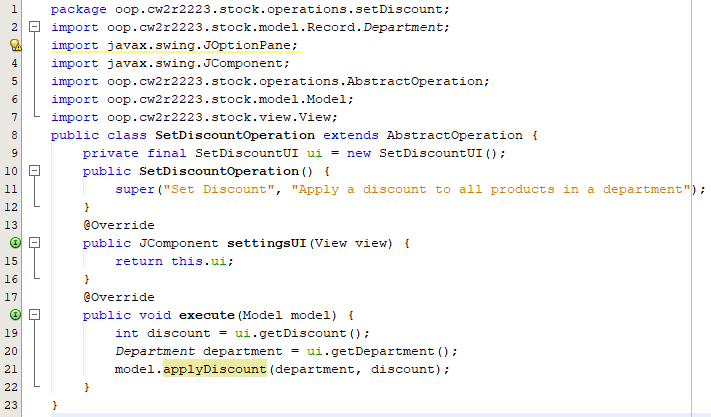


Figure : SetDiscountOperation code



Figure : SetDiscountUI

# **References**

[1] “The JComponent Class (The JavaTM Tutorials > Creating a GUI With Swing > Using Swing Components).” https://docs.oracle.com/javase/tutorial/uiswing/components/jcomponent.html (accessed Mar. 29, 2023).

[2] “Lesson: Getting Started with Swing (The JavaTM Tutorials > Creating a GUI With Swing).” https://docs.oracle.com/javase/tutorial/uiswing/start/index.html (accessed Mar. 29, 2023).

[3] “JPanel (Java Platform SE 7 ).” https://docs.oracle.com/javase/7/docs/api/javax/swing/JPanel.html (accessed Mar. 29, 2023).