Reflective Report on the Fast Courier Service GUI

1. Analysis of the product

Package Structure and Libraries

The graphical user interface for this assignment was placed into a new project. This was done instead of placing the GUI into the data model package because this keeps the front end separate from the back end. The data model is also an entity that could be reused in future projects for a different delivery tracking company.

This project has four libraries attached to it which include the Fast Courier Service data model, the observer pattern library, the utilities library which includes the command pattern and the JCalendar library. Figure 1 illustrates the dependencies clearly. This package hierarchy follows in accordance with the class design principles such as the Common Closure Principle, the Stable Dependencies Principle and the Stable Abstractions Principle. The reason they are in line with these principles is because no longer does a change to the GUI force the release of all the other packages, all four were in one package. The dependencies between the packages are now in the direction of the stability of packages. The more stable packages are also much more abstract which is demonstrated by the GUI only having one interface and it is not depended on by any package. The other packages have more than one abstraction and so the Fast Courier Service data model depends on them. The Fast Courier Service data model also depends on the observer pattern which has more abstractions than itself.

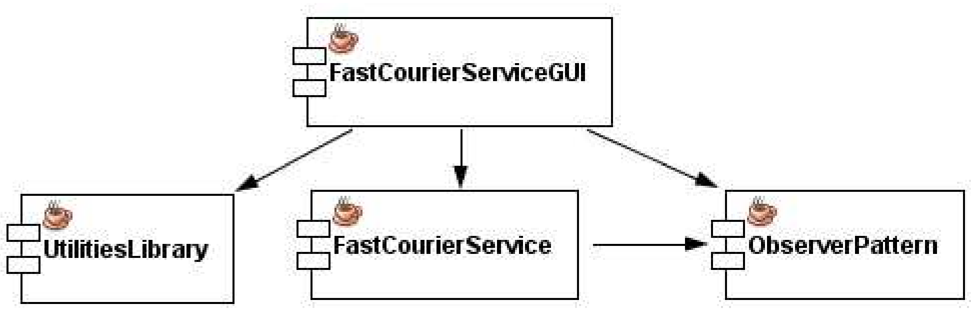

Figure 1. The dependency tree.

The structure of the graphical user interface project has two source packages, one for all the GUI components and one for the specific command pattern classes created to facilitate redoing and undoing of certain actions.

Classes

In order to minimise the amount of code written and to follow 'DRY' coding, some of the classes are used reused for differing purposes such as the AddEditCustomer class. Since adding and editing a customer would require the same input fields it was decided that one class should be created for both and then an enumeration is created which will be used to declare what the window will be used for.

This approach was used for viewing customer deliveries too. Instead of viewing all customer deliveries together, it was deemed better to view incomplete deliveries separate to complete deliveries as this will save a fair amount of time when looking for a specific undelivered or delivered delivery.

For the three reports, again, only one class was created and an enumeration is used to declare what type of report it will be during runtime.

All javadocs have been generated for libraries created such as the fast courier service data model, the observer pattern and the utilities library.

Tests

The data model was altered to facilitate using the command pattern. JUnit tests were run to make sure that the data model was still functioning correctly and all 72 tests passed successfully.

Project Analysis

All of the requirements specified in the document have successfully been implemented. Following will be an explanation of where additional functionality has been implemented.

The consignment number has been made to automatically create the next number in the sequence whenever a new delivery is created for a customer. This approach saves the user a lot of time as they no longer need to find the next number in the sequence themselves. This also eliminates the chance of an error such as duplicated consignment numbers or the user entering an incorrect number by accident.

The command pattern library was attached to the project and commands written so that user actions could be undone. This gives the user the feeling that they are in more control of the application and a great sense of security if something went wrong. A separate package was created, within the GUI project, to house the commands specific to this user interface project. Four commands were written which allow the user to add or undo adding a customer, add a delivery to a customer or undo that action, edit a customer’s details or reverse the editing and finally updating a status of a delivery can be undone.

Whenever an action is undone or redone a message will appear at the bottom of the main window of the application. The reason for this is that good user interface design dictates that every action a user performs should provide immediate feedback. Just by clicking undone in the edit menu will not always provide feedback since a new window will need to be opened to see the changes such as reversing the updating of a status. Therefore the message is a means to give the user satisfaction that what they have just done has been done successfully.

A further addition was the implementation of multithreaded reports. If this company had a very large number of customers then every time an update to the data model occurred the open reports would need to refresh the information and this could potentially take a few seconds. As a guiding rule, anything that takes longer than a second should be placed in its own thread. For this reason the reports are multithreaded.

How the multithreading works is as follows. Whenever the update on the report is called, a new PopulateReport is created. This is a class which extends SwingWorker and implements IPopulateReport. IPopulateReport is an interface that only provides the populateReport method. Once a new PopulateReport is created then the public method ‘populateReport’ is called. This allows all the report’s table information to be retrieved in a separate thread in the background so that the main application window can continue to be used.

Once the ‘populateReport’ method is called, this then calls ‘execute’ which allows the table data to be fetched and published row by row to the report window. Once all rows have been fetched then the total cost of all deliveries is summed and displayed. If no rows are fetched due to filtering out customers without deliveries, then the table will be empty except for the total being displayed as zero on the bottom.

Other functionality that has been added for the reports is the ability to filter the results of a report by statuses available for each report. For example, since the completed deliveries report only shows delivered and undeliverable deliveries then that report can only be filtered using those two statuses.

Another filter feature is the ability to remove any customers from the results who currently do not have any deliveries. These filters make the results much more compact and easier to make sense of.

Faults Found

For an unknown reason whenever the ‘Save’ button is clicked when updating a status and then ‘No’ is clicked when asked whether you would like to save, the dialog window’s width shrinks to about half of what it normally is. Attempts were made to manually set the width to the required width but it continued to do as it had done before.

With the multithreading, whenever all three reports are opened at the same time one of the reports seems to fetch all its data instantly whereas the others take the required time set by the loops within the class which are used to test the multithreading. If the windows are opened separately then they all take the right amount of time to fetch data. This only occurs when the method which contains the loops is uncommented within the ‘PopulateReport’ class.

2. Analysis of the approach

The work on this project was started as soon as possible so that the core functionality could be in place long before the deadline. That way more bugs could be found and removed as well as giving time for extra functionality to be added such as the command pattern and multithreading the reports windows.

Conducting a project such as this has been very beneficial in terms of understanding in more detail the use of design patterns as well as creating a graphical user interface that uses multiple JFrames and modal JDialogs that have text fields and buttons within them. A project such as this has never been attempted before. A project which uses this many libraries, has so many windows all linking together in some way and using a variety of design patterns which include the observer pattern, command pattern and decorator pattern.

The knowledge that has been gained will undoubtedly continue to be used throughout the career that lies ahead as a software engineer. Having an understanding of the principles of good object-oriented design has helped tremendously in areas such as ensuring packages are not dependent on packages less stable than themselves and separating classes into packages where they work as one unit. An example of this was that data model contained the observer pattern with it in the same package and at the start the graphical user interface was created within the data model package. This went against so many design principles and therefore they were all separated into independent packages.

Overall the feeling is that this project has been a success even if only for expanding knowledge and gaining experience as a more competent software engineer.