Description

The program designed is a basic vehicle ordering system where a user will perform some minor configurations on a vehicle they would like to order and subsequently submit that order.

Once can also ‘Dealer’ persona where, as a dealer a user is able to fulfill that order which is a virtual representation of actually having that vehicle delivered or in some other manner completing the transaction. An order consists of a user name, a trim level as well as the extra options a user may or may not choose to purchase. This order is then stored and can be reviewed at a later time. It should also be noted that trim levels differ with respect to their base makeup, which in turn allows the base price to be different and as trim packages change the options that may be added to a trim package also differ. Similar to some extent to a real world car buying experience as we know it now. The price of the base vehicle and the options are also noted and differ depending on the various circumstances and of course the options themselves.

Golden Path

The Golden path for this application is for a user to create in order from start to finish all the way through a submission. Next to restart the program and assume the ‘dealer’ persona and the order to be marked as fulfilled. One could easily see, given the program description and this Golden path that there are multiple possible program branches that are available.

Patterns

The patterns that were used were are the iterator pattern, the proxy pattern, state design pattern as well as the strategy design pattern.

The **iterator** pattern was used to iterate over the standard features of a particular trim and to populate the GUI. This was done using a prebuilt array that was iterated over in order to populate the various GUI fields. This allowed each trim package to merely consist of a list and allowing the iterator to pull item by item and feed the GUI properly with the structure itself handling the size of the list and ensuring that each item on the list would be retrieved.

A **proxy** pattern was used in order to change the access levels as a dealer and a customer in general were allowed to do different things, and this proxy handled. The permissions given the type of user that currently using the system. For instance, when a dealer is selected, they are not able to create an order, but rather they must retrieve a previously created order and from there can review and fulfill the order, as opposed to make changes to the order. This is in contrast to a customer that may recall an order and make changes to it at any time, however, a customer is not able to fulfill an order, as would a dealer be able to.

A **state** design pattern was used in order to regulate the status of a particular order. The statuses (not in any particular order, are “editing” “canceled” and “submitted.” These states used in the state pattern coupled with the proxy pattern regulates the flow with respect to state transitions as not only can these be dictated in a binary method (allowable or not, depending on the current user) but the current state dictates the states to and from which transitions may occur.

Also used was a **strategy** pattern and this was used in order to allow a particular trim to be easily altered. A trim variable was assigned, and as the user changed the trim level. The trim variable was changed to a different trim class that had different attributes associated with it. However, dynamically while a user was able to change a trim class on-the-fly there is no need to have anything predetermined or more importantly invoke an instance of a particular trim class on-the-fly, but rather set the trim class as needed, which is the power of this particular design pattern.

Testing

Overall coverage achieved from testing the code base is approximately 33%. The patterns that were completely and successfully tested are the: proxy design pattern, iterator design pattern, and the state design pattern. As the strategy design pattern entirely resides in a single Java file. JUNIT testing would not access any of the contained classes. An attempt to make each of these classes public prompted the error that the classes themselves required them to be moved to another file before being made public, and as the application worked as desired given the current set up, I thought that it would lack integrity to restructure the code to satisfy a unit test. Therefore, I was unable to implement unit tests for the strategy design pattern. Although I believe it was not part of the assignment my goal for unit testing was to achieve 100 percent coverage. However, as I set many functions as void, as there was no real reason to receive a return value, those could not directly be examined, and as many of these void functions called other void type functions. It was very difficult to find that proper side effect that a test would need to validate.

Another testing issue was that of the GUI. Any attempt to test any of the GUI files, ultimately crashed as items were invoked from the \*.fxml files. Also, GUI items were referenced so unit tests would not necessarily present these GUI items required for interaction; while there may be a method to perform such unit tests I am not aware of it, and in general feel it may be beyond the scope of the assignment.

Technical Description

The GUI for this application was compiled using Java FX runtime version 8. However, I received warnings that a Java FX API version of 11 was being loaded. Nonetheless, this seemed to not be an issue that as far as I can see with regard to the running of the application itself. It should also be noted that this application was developed and tested on a Windows machine. Now, while Java is portable. I am not sure how this will run on a Mac or a Linux machine. I don't believe that any of my file I/O specifically references any Windows specific items but I can’t be sure. Conversely, I at the current time don't have a Mac OS try this out on nor the bandwidth to spin up a Linux machine to run this application under.

As far as storing and re-calling the orders, a JSON is used and updated as needed. Therefore, parser classes needed to be developed in order to retrieve various pieces of information from the JSON for use. Another issue that I ran into with this was a section of JSON, may have not been populated not due to any error but rather during the normal flow of the program's normal running it was not required. Therefore, I needed to still provide fields that may remain unpopulated and create unpopulated fields and some as the JSON is stored in a file ,recall at a later date. I developed a rudimentary file handling class in order to create a file for use, and to later store the information to a file. Also, as this file may contain multiple orders, I not only needed to parse JSONs, I also needed to store orders in a manner that I could retrieve orders one by one from the JSON file, make any changes, and then update that JSON file for subsequent recalls. Therefore, a good deal of checks were employed to ensure that the proper things were done at the proper time.

Problems:

I am more familiar with Swing however, I'm under the impression that it is an older way of doing things and that JavaFX is a more modern paradigm. Accordingly, I decided to stretch myself and attempt this project in JavaFX. I did locate a GUI builder as I don't believe it is efficient to building a GUI in code. I see no issue with establishing a GUI device or control in code. However, the physical placement of said device within code seems to be more tedious than it's worth.

That being said, placing the items using the various layouts is something that I am not very well-versed in therefore, the graphic design of this application has room for improvement. Another issue that I was grappling with upon design of this program is that the size of a particular window is dictated before it is called as opposed to invoking a particular window and the window itself knows its size. So while I designed a particular window with a particular orientation; upon its calling, those dimensions may have changed and sometimes the dimensions are inherited from places I have yet to find. I spent a lot of time attempting to have a GUI launch various things, and perform specific set of activities upon its instantiation. It was only later in the project that I found there is an ‘initialization’ method that could be used. This is an important detail as when a stage is launched not all of the controls that reside on that stage have been instantiated. Therefore, I would run into a great deal of problems upon launching a stage and having the program crash as a particular control is simply not available. What I needed was something analogous to “onload(). ” Meeting after everything has completed loading then continue with the initial GUI setup. I did try a great deal of trial and error and in some places it did work. However, other places I would end up initializing something and performing the same “work around” again and reinitialize things. For instance, I used a “mouse over” GUI routine as my initializing function. Therefore, if the mouse were to leave the window and return it would reinitialize everything again.

Oddly enough, the FXML files invoke the constructor of their controlling class. However, if I put items within the constructor that referred to GUI controller items the program would crash and this was proof that while a constructor is called it would still be too early to initialize or refer to various GUI items. This is a very frustrating, tricky situation until I found the "initialize" method which was exactly what I was looking for.

Another big issue I ran into was the storing and retrieving of information. I spent a great deal of time attempting to work with XML files. However, I could not consistently update nore retrieve items, especially with multiple levels for one particular item / record. I attempted the use of various packages and had limited success and almost importantly had a limited amount of time. Therefore, I move over JSON files, which are relatively flat and serve the purpose that I needed.

Coverage

Initially, as I was attempting to gain hundred percent coverage for my JUnit tests. I began to remove unused code, however, some of these code items. I removed were unused helper functions that while useful were unused in this project. Theses items were slated for use but were made OBE for some reason or another. After some editing I came to the conclusion that this was not a constructive effort or best use of time. Therefore, I decided to not clean up the code; true this is not a best practice, but there have been times on this particular project where I would remove a particular code snippet and after testing would find that I needed that snippet, but slightly different and had to spend the time re-creating it. Therefore, I in general like to keep old unused commented out code around as I'm never sure when I'll need to refer back to it. While it hurts readability that is the only drawback I see as in general as it is not compiled in so this size of the application is minimally affected if it all (assuming I'm not compiled this is a library) and it's great to have that functionality still there in case there is a need to alter the application in a manner that will use it.

Final Thoughts

The GUI is not fancy, nor is it one hundred percent consistent, but I did stretch myself by using a different GUI development package and although not employed in the final project, delved into the world of XML. This also allowed me to learn a good bit about XML and have hands-on experience with it. I have also incorporated some items on the GUI, but for whatever reason, they do not seem to appear. Another interesting aspect was some of the design patterns are directly incorporated into managing the GUI with respect to allowing user specific actions and / or showing or hiding various features of the GUI. This made it apparent to me that at the very least design patterns are not only for back in decision-making. Still, while implementing a front end they are as useful if not more to regulate program flow. That's one reason why I believe that these design patterns should be part of a programming 101 type of academic program as these are very useful concepts that that times, myself included, have been informally and potentially improperly implemented. Due to their usefulness the fact that these design patterns formally exist and have been flushed out, in the realm of many more senior programmers these are hidden and potentially unknown gems with regard to object-oriented programming.

I believe that this was to applicable assignment not only to the course, but also with respect to the real world in a real production environment. My attempt to use XML burned a great deal of time and reminded me why I personally really use it for storage for the. I do concede that a proper storage mechanism would be a database, but that will involve a user of this program to install and configure such an item and that was well beyond the scope of this particular assignment. Also, the GUI aspect of this project took a great deal of time as well because I was not familiar with the launching is initialization of a GUI and it was unclear to me until a lot of trial and error with respect to when and how things occur.