Problem Domain: Interactive Garage Application

This version of the garage code is based on the work in project 2 and 3 but will allow an outside user to interact with the garage in the role of the Mechanic.

Application Requirements

The program must function as follows:

The system will initialize the same types of vehicles used in project 2 or 3, but there will only be 1 instance of each sub-subclass (Bike, SUV, etc.). The GarageAnnouncer is not used in this application. The application will only run for a single day, from 9 AM up to 8 PM (if not ended sooner).

The system will welcome the user to the application "A Day at the Garage", ask them for their name, and begin at the simulated opening time of 9 AM.

At each hour, the user may attempt to perform one of the Mechanic's tasks: Unlock all vehicles, Wash all vehicles, Tune-up all vehicles, Test Drive all vehicles, Lock all vehicles. In addition, the user can select Leave for the Day, which will end the application at that hour. You may wish to number these options to make it easy for the user to enter their choice.

Wash, Tune-up, Test Drive, and Lock tasks should not be allowed if vehicles have not been previously Unlocked. In such cases, the user should be informed of the error, and allowed to select another task.

The results of choosing a valid task will be much like the project 2/3 simulation, applying the task to the vehicle should cause a status print message and the vehicle should respond with its status:

> "Bruce unlocking SUV S45T4D. SUV S45T4D is unlocked."

The outcome of most activities should be the same as in Project 2 and 3 (possible crashes, sparkle vs. shine, sputter vs. run). The Wash activity will be modified to have variation for some vehicles, and each of these algorithm steps should provide their own output. Variations are as follows.

> Normal vehicle Wash activity will have three steps: Soap, Scrub, Rinse, Dry
>
> Monster trucks will Wash as follows: Soap, Scrub, Rinse, Dry, Wax, Wax, Polish, Detail
>
> Convertible cars will Wash as follows: Soap, Scrub, Rinse, Dry, Wax, Detail, Detail
>
> Output for a typical vehicle being washed would look like:
>
> "Bruce washing SUV S45T4D. SUV S45T4D is Soaped. SUV S45T4D is Scrubbed. SUV S45T4D is Rinsed. SUV S45T4D is Dried. SUV S45T4D sparkles."

If the user crashes the Monster truck in a Test Drive, the system should end (with an appropriate message).

All user interactions will be via a console or terminal text-based command line interface. Input and output does not have to be programmatically stored in a text file, sessions can instead be captured via copy/paste.

Development Requirements

You must use an application of the Command pattern to provide the structure for interacting with users to cause actions to occur on Vehicles.

You must use an application of the Decorator pattern to extend the Wash activities for Vehicles not using the basic Wash set of four methods.

All interaction with the user from any class should use a Singleton object, instantiated by its first use. This object will provide methods for getting user input and printing output messages.

Code where these patterns are implemented should be clearly commented for ease of identification. You may continue to use other patterns in your code as warranted by your design.

Test Requirement

Include at least 10 JUnit test methods (methods with the @Test decorator) in a single test class called MyUnitTest. These unit tests should be applied appropriately to classes and methods in your simulation code to ensure their proper behavior. You should be able to instantiate a MyUnitTest object before your simulation run and then execute each of the ten test methods at some point before, during, or after the simulation, with output from each test indicating the identification of the test and success or failure of each test method. A single test method may contain single or multiple assert statements as you feel are needed or are appropriate. Clear output from the test run must be included in your program results.

I suggest using a simple test class containing test methods using a variety of assert statements as outlined in:

- http://tutorials.jenkov.com/java-unit-testing/simple-test.html
- http://tutorials.jenkov.com/java-unit-testing/asserts.html
- http://tutorials.jenkov.com/java-unit-testing/matchers.html

I suggest using either JUnit 4 or 5, but that is up to you. If you or your team are more experienced with JUnit, you may want to implement the tests in a more sophisticated fashion. As long as at least ten test methods are created and clearly applied, you may use any alternate implementation approach.

Extra Credit

For up to 10 points of extra credit, apply the Template pattern to the basic Wash behavior to vary the implementation of the Soap, Scrub, Rinse, Dry base methods based on the subclass (Car, Truck, etc.) of a Vehicle being washed. In defining the Wash algorithm, make at least one of the methods abstract (meaning the subclassed version of the Wash algorithm will need to implement the method), make at least one concrete (implement the method in the Wash algorithm and do not allow subclassed algorithms to change it), and make at least one a Hook method (the Wash algorithm will provide a base implementation, but the subclass may override it). Be sure that the behavior of the modified methods result in visible text messages that show the subclassed algorithms are being used by the subclasses of Vehicles. Clearly comment in the code where Template is applied, and ensure the three method variations (abstract, concrete, and Hook) are identified in the code.

Grading Rubric

As always, your OO code should be well structured and commented, and citations and/or URLs should be present for any code taken from external sources. Comments should focus on what is being done in the code, not on how, unless the method in question is unclear or obscure. You may not directly use code developed by other student teams, although you may discuss approaches to the problems, and may wish to credit other teams (or class staff) for ideas or direction.

The submission is a GitHub Repository URL. The Repo must contain:

- Running, commented OO code for the simulation in Java 8 or later
- A README Markdown document including project title, project team names, any comments or assumptions made for development, any issues encountered during development, and any special instructions to run the application; use Markdown syntax to make the file more readable, including use of section headers.
- A text output file containing a capture of at least 3 interaction sessions with the program demonstrating input and output text messages that illustrate the code works as described above. You can simply copy your interactions with a command line in a console or terminal and paste them into an example output text file.
- A PDF containing a full UML Class diagram that shows classes and relationships from your design (include key attributes and methods in the class diagram, as well as pattern identification)

Homework/Project 4 is due at 8 PM on Wednesday 6/30 and is worth **75 points**. There is also an additional **10 point** extra credit opportunity (for Template as described above).

- 10 Points – README file (-1 or -2 for incomplete or missing elements)
- 10 Points – PDF with UML class diagram (-1 to -2 for poor quality elements or lack of pattern identification, -10 for missing UML diagram)
- 15 Points – Use of 3 required patterns: Singleton, Command, Decorator (-1 or -2 for implementation issues or poor commenting per pattern, -5 if a pattern is missing)
- 10 Points – Implementation of 10 JUnit tests as described above, including captured output from test execution (-1 or -2 for poorly implemented tests, -5 if test output is not shown, -10 if not implemented.)
- 15 points – Execution as evidenced by text-based output file (-2 or -4 for missing expected functionality or output elements)
- 15 Points – Code quality (-1, -2, -3 for issues including poor commenting, procedural style code, or logic errors; -15 if not an object-oriented solution as requested).

Any UML tool can be used to create the UML class diagram. If done on paper/pencil or whiteboard, please be sure the diagrams are readable and clear for grading.

Assignments will be accepted late as follows. There is no late penalty within 4 hours of the due date/time. In the next 48 hours, the penalty for a late submission is 5%. In the next 48 hours after that the late penalty increases to 15% of the grade. After that point, assignments will not be accepted.

Late penalties will be waived for health or medical related issues (of course). Use office hours, e-mail, or Piazza to reach the class staff regarding homework/project questions.