1. Recall from Project 2 the OO program you your team wrote in Java 8 or later that implements a Garage full of Vehicles. Using that code and its original requirements as a basis (see the previous assignment details if needed), extend your Java code to perform the following additional tasks:

a) Add an implementation of the Factory pattern for code that creates Vehicles of a requested type. The Vehicles will vary by their low-level subclasses (Wagon, SUV, etc.), the factories will vary by the mid-level type (Car, Truck, etc.) so we should see concrete factories similar to CarFactory, TruckFactory, etc. The only place a new operation for a Vehicle subclass should be called is within the Factory code. Clearly document the Factory code.

b) Add at least one implementation of the Strategy pattern by having at least one Vehicle behavior delegated and referenced by Vehicles rather than being inherited and overridden. This means when Vehicle subclasses are instantiated, the behavior they need will have to be initialized for them in a Strategy pattern manner. Clearly document in the code where the Strategy pattern is applied. You may use Strategy more than once if your design calls for it.

c) Add an implementation of the Observer pattern. Create a new observer class called the GarageAnnouncer. At the start of the program, the GarageAnnouncer (who IS-A GarageEmployee) will arrive at the Garage each day. The GarageAnnouncer will observe the Mechanic, which will be modified to be observable. When the Mechanic is starting to perform one of their tasks, they will create an observable event for the GarageAnnouncer. The GarageAnnouncer will respond to this by announcing the event (issuing a print statement) something like “Hi, this is the Garage Announcer. The Mechanic is about to wash the vehicles!”. (Note, this should not replace the text messages the Mechanic issues from the original flow.) Once there are no further events to announce for the day, the GarageAnnouncer should announce the Garage is closing and cease observing and leave for the day. Clearly document in the code where the observer pattern is applied. You may use any valid Java coding approach for observer implementation – write your own observer/observable interfaces or use the java.util.concurrent Flow class or use the java.beans PropertyChangeListener/PropertyChangeSupport.

d) Add an object to track time at the Garage called a GarageClock (the GarageClock class is NOT a GarageEmployee). The Garage will be open from 8 AM to 8 PM each day, and the GarageClock will maintain the current time increment. The GarageClock will also be an observable for the GarageAnnouncer, who should announce every time change from 8 AM to 8 PM. The Mechanic must perform their tasks (unlock, wash, tune up, test drive, and lockup) for all vehicles once per day at a time of your choosing. For instance, if the clock changes to 9 AM, you might choose that as the time the Mechanic will unlock the Vehicles.

Code should be clearly designed in an OO fashion throughout and should be appropiately documented. Full output for a ten day simulated run from the program should be captured and provided as in Project 2. Clearly identify the source of messages in their text: Vehicles, Mechanic, GarageAnnouncer

2. Create a complete UML Class diagram for the revised Garage Program with Factory, Strategy, and Observer. The Class diagram should be detailed, with all attributes, accessibility, and methods documented, as well as any requirements for multiplicity in associations (e.g. two of each Vehicle subclass, etc.). This must be created by hand, not automatically from code. Highlight the pattern implementations in the diagram.

3. Create a complete UML Sequence diagram for a typical day for the revised Garage Program. This diagram should show key top-level objects (:GarageAnnouncer, :Mechanic, :GarageClock, :Vehicle) that are instantiated, executed, and deconstructed in the lifetime of the code. To make this readable, do not go to the subclass level; so Vehicle, yes, but probably not Car or Convertible) and the interactions between those objects. You can also just diagram the "happy path" of normal operation without exceptions.

4. Create a complete UML Activity diagram for the revised Garage Program, showing all major operations that occur from the beginning to the end of code execution for a single simulated day. (Again, details at the subclass levels are not required.)

5. Create a UML Use Case diagram for the tasks the Mechanic (the Actor) must perform at the Garage. Consider any <include> (mandatory/ancillary) or <extend> (optional/conditional) tasks they may need to include in a complete view of the scenario. Use the WAVE rule to guide your use case model.

There is a 5 point extra credit element available for this assignment. For extra credit, import a version of JUnit, and use at least five JUnit test statements to verify some of your starting expected objects are instantiated. Use a command line argument to be able to run your code with or without tests, and capture a version of your output that shows how the output differs when the tests are run.

Homework/Project 3 is worth 75 points – 40 points for the program in item number 1, 10 points each for the UML diagrams in 2, 3, 4; 5 points for UML diagram 5. Again, a potential additional 5 point bonus is available for JUnit integration as described.

Grading Rubric:

Question 1: 40 points possible.

- README Documentation 5 points (-1 or -2 for incomplete or missing elements, -5 if not there).
- Execution 10 points (-2 or -4 for missing expected functionality or output).
- Code quality 10 points (-1, -2, -3 for issues including poor commenting, procedural style code, or logic errors; -10 if not an object-oriented solution as requested).
- Proper use of factory, strategy, and observer patterns 15 points (-1 or -2 for implementation issues or poor commenting, -5 if a pattern is missing).

Any UML tool (draw.io/diagrams.net, etc.) can be used to answer questions 2-5. If done on paper/pencil or whiteboard, please be sure the captured diagrams are readable and clear.

Questions 2-4 will be graded on the completeness of the answer, and the correct use of UML. A solid answer will get 10 points, significant missing elements or mistakes in UML answers will cost -2 points each, missing parts of answers completely will be -4 points.

Similarly, Question 5 (which should be a simpler answer than 2-4), is worth 5 points, with -1 or -2 for missing elements or severe mistakes in UML.

There is a 5 point bonus if the JUnit functionality is included as described, including example output and README documentation of how to change the command line to run the code with or without tests.

Submissions:

Question 1 (the application) should be submitted via a GitHub Repository URL, including the code, the output text file, and a README Markdown file with the title of the project, the names of team members**,** any comments on or assumptions made for the development, and any special instructions to run the code. 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.

Questions 2-5 (UML Diagrams) must be submitted in a single PDF file, must include all names of team members, and should be included in the project repo**.**

Homework/Project 3 is Due at 8 PM on Wednesday 6/23.

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.