CS4354 – Fall 2014 – Assignment 4

Due date: Wednesday, Nov. 5, 2014 at 12:00 noon.

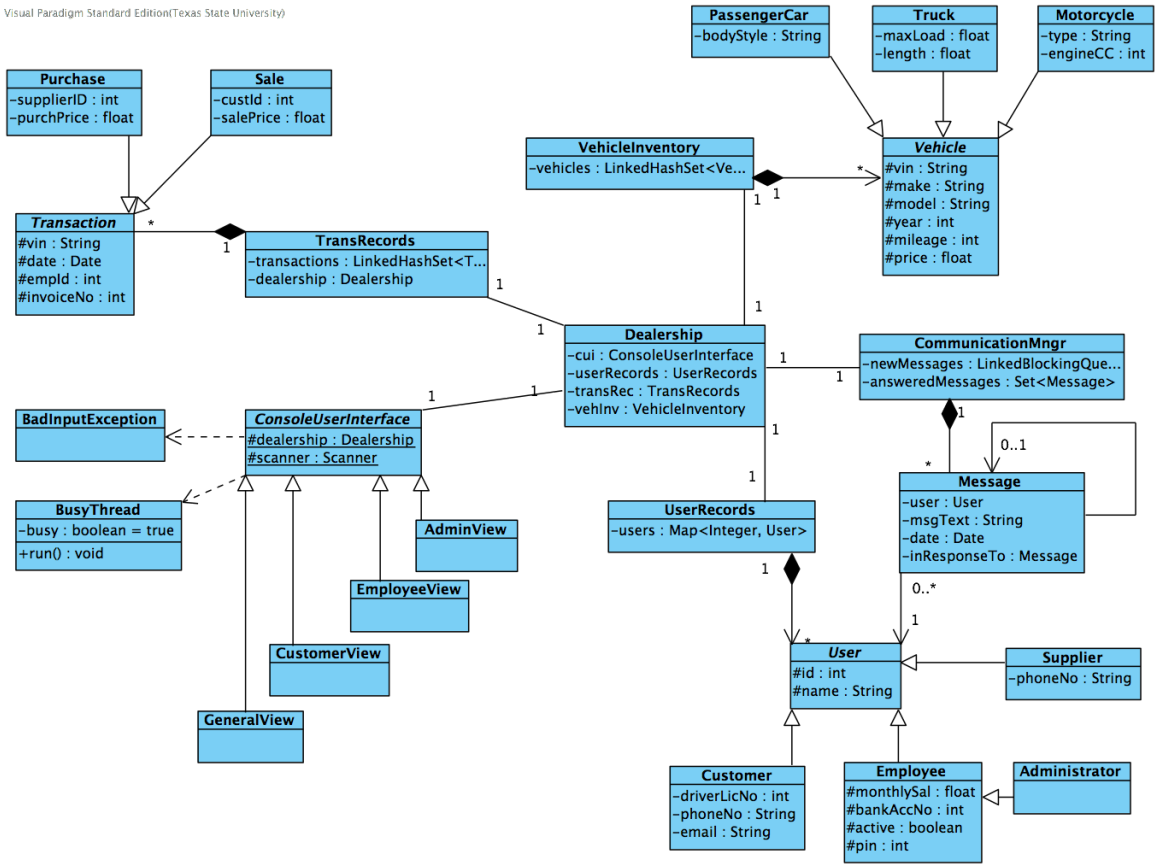
Goal:

The goal of this assignment is to help students better understand the following concepts:

1. Implementation of Java code from a given UML Class Diagram
2. Use of Java Collections
3. Use of Java Logging
4. Use of Java Threads

Description:

This assignment builds on the car dealership software from the previous assignments, adding some extra functionality and using Object Oriented Design Patterns to achieve better structured and more maintainable code. The image bellow shows an overview of the UML Class Diagram of the software:



The above class diagram is also provided as a separate Visual Paradigm project file for convenience. The students can use the Code Engineering tool of Visual Paradigm to directly generate Java code from the given class diagram. The generated code will contain all the required classes, most of the class attributes, and the declaration of most of the basic class operations (methods). The student will have to add the missing code to create a fully functional application.

Compared to the description of *Assignment 3*, the accounting management part was left out for convenience, and the customer search and communication options are implemented as command line operations, as opposed to a Web interface.

Specifics:

- **Interface**: As it can be seen from the class diagram, the user interface has now been decoupled from the rest of the application. This is a standard practice in application development, by which the front-end of the application is kept as independent as possible from the back-end. The front-end handles all the user input/output whereas the back-end deals with the data management. In this case, the object of the classes `CustomerView`, `EmployeeView` and `AdminView` act as boundary objects and communicate with Controller objects such as `TransRecords`, `CommunicationMngr`, etc., to complete the necessary operations. The front-end collects all the necessary input from the user and then calls a method of the back-end to perform an operation, passing the collected input as arguments. The result is returned back to the front-end and is subsequently output to the user. When the program first starts, it loads the GeneralView. From there the user is shown a menu and can chose to switch to the `CustomerView`, `EmployeeView` or `AdminView`. To switch to employee or admin view, the system asks the user to enter their PIN. The fist admin user of the system has a standard PIN that is known by the system and the user. For every other admin or employee added to the system, a new PIN should be entered.

- **Collections**: This implementation makes use of different Java Collections to store the data related to the application. Each collection type was chosen based on its functionality, while taking into consideration the time complexity of each operation that will be performed on the data. For example, for the collection that stores the completed transactions, we don't want to allow duplicate transactions but care about the order in which the transactions were completed, hence, we opted in using a `LinkedHashSet` collection type. The students should be able to understand why each collection type was chosen, and also be able to use the necessary operations of each collection to add, retrieve and update data.

- **Threading**: In order to keep the application responsive, it is a common practice that the front-end (graphical user interface) runs on a different thread compared to the back-end. That way, if a back-end operation (e.g.

searching for a vehicle) takes a while to complete, the front end does not seem frozen and can still respond to the user. Since the amount of data in our application is small, our back-end operations normally complete instantaneously. To simulate the case where a back end operation might take a while to complete, we can artificially slow down some operations. For example, let's take the method `searchInventory()` of the class `VehicleInventory`. To slow down the operation we can pause the execution of the method for a random number of seconds, say from 0 to 10. We can do that by adding the following code in the body of the method: `Thread.sleep((int)(Math.random()*10000));` In order to keep the user informed that application is still running, the interface can show a progress bar. We can do that by starting a separate Thread, which prints out a text-based progress bar. Right before the call of the back-end operation, the interface can start the thread and set the `boolean` variable "`busy`" to `true`. As soon as the back-end operation returns, the variable `busy` can be set to `false`, thus stopping the execution of the progress-bar thread. The code bellow shows an example of what such a thread could like:

```
public class BusyThread extends Thread {
    private boolean busy = true;

    public boolean isBusy() { return busy; }
    public void setBusy(boolean busy) { this.busy = busy; }

    public void run() {
        System.out.print("\nWating for response.");
        while (busy) {
            try {
                System.out.print(".");
                Thread.sleep(500);
            } catch (InterruptedException ex) {
                //Interrupted
            }
        }
        System.out.println();
    }
}
```

- **Logging**: To keep track of possible errors during execution, applications usually log these errors to files and also display some of the errors to the user. Java provides such a mechanism for data logging. For this application, you are asked to use the Java Logging Framework to log all possible errors and exceptions that may occur in your program. The errors should be logged into a file and the console at the same time.

- **Communication**: Communication is a new functionality compared to Assignment 2. Its purpose is to allow a customer to send a question message to the dealership and receive a response. When the customer sends a message, that message is added at the tail of the `newMessages` queue of the `CommunicationMngr` class. An employee has the option to check the queue

for unanswered messages and answer one at a time. Messages are extracted only from the head of the queue. Each customer message that was answered is moved to the `answeredMessages` set. The employee response message is also added to the same set.

Tasks:

1. Use Visual Paradigm to generate Java code from the given class diagram and implement all the operations described above.

2. Try to make your program as robust as possible, by using Exception handling to deal with possible problems that may occur during the program execution.

3. Use a standard Java coding style to improve your program's visual appearance and make it more readable. I suggest the Google Java coding style: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

4. Use Javadoc is for every public class, and every public or protected member of such a class.
   Other classes and members still have Javadoc as needed. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

Logistics:

This assignment will be done and submitted **individually** by each student.
Submit your answer in a single file (assign4_xxxxxx.zip). The xxxxxx is your TX State NetID.

Submit an electronic copy only, using the Assignments tool on the TRACS website for this class. Do NOT include executable or .class files in your submission.