COIS 2240H – Software Design & Modelling

Assignment 3 — UML Design and Modelling

Total Points: 100 – Due April 4th, 2025

Note: This is a **group-based assignment**, allowing you to work in groups of a maximum of 3 students. While **individual submissions** are permitted, they **are not encouraged**. Any student who did not contribute to completing the assignment tasks must not be included in the submission, and should be reported to the instructor to be removed from the group. Any issues related to academic integrity, such as plagiarism, copying from another group, or unauthorized use of generative AI tools will be handled in accordance with the university's academic integrity policies.

Deadline: 10% will be deducted for every day of late submission, and no more than 5 days late. If a second attempt is submitted after the due date, the submission will be considered late.

Objectives:

Extend and improve an existing software system with design patterns, test it to ensure quality and reliability through unit testing, and maintain proper versioning by utilizing version control.

Introduction:

For this task, you will demonstrate your ability to extend, maintain, and test an existing software system, following standard design and testing methodologies. Specifically, you are expected to:

1. Extend a software system based on given specifications.
2. Implement proper file storage and retrieval for system data.
3. Test the system to ensure functionality, reliability, and performance.
4. Refactor software source code to address certain coding choices
5. Use version control (Git) to manage and track software versioning throughout the development process.
6. (Optional/Bonus) Implement a Graphical User Interface (GUI) for data entry and display.

To successfully complete this assessment, you will need to undertake the following tasks.

Task 0: GitHub Repository and Commits [20 Points]

It is expected that you will have at least **11** commits, representing milestones for the work completed in the following *Task 1* and *Task 2*. You may have additional commits to show efforts in fixing errors, reorganizing, or documenting the code. This will help you practice code versioning, which is important for managing code changes, tracking progress, and troubleshooting. All group members should collaborate and contribute to committing to GitHub through their GitHub accounts.

Submission:

1. Create a new *GitHub* repository called ***COIS-2240-Assignment3*** or fork it from this repo *https://github.com/taher-ghaleb/COIS-2240-Assignment3*, then clone it onto your computer.
2. You may use *Git* command line or *GitHub Desktop* to handle Git operations.
3. Create an *Eclipse* project called ***RentalProject***, either inside the cloned repository or in a separate folder, but ensure you still commit changes regularly after completing each subtask and push them to the GitHub repository.

4. Copy the assignment source files into your Eclipse project.
5. The first commit should contain the original source code files.
6. In addition to the screenshots showing your commits for *Task 1* and *Task 2*, make sure your *GitHub* repository is ***Public***, and **share** its **URL** in the submission comment **on Blackboard**.

Task 1: Software Extension [50 Points]

The code provided is for the Vehicle Rental System you worked on in previous assignments. The system currently involves 11 classes. You are required to extend this software by implementing file-based storage and a user interface for interacting with the system. Run the program and follow the menu options to add vehicles, add customers, rent a vehicle, return a vehicle, etc. **Note** that, after completing **each** of the following subtasks, you will need to *submit a Git commit* containing the changes and add a proper commit message describing the changes made.

1. **[**9 Points**]** Refactor the **RentalSystem** class to follow the **Singleton** design pattern, ensuring only one instance of the class exists throughout the application's lifecycle. Introduce a getInstance() method that returns the single instance. Update the **RentalSystemApp** class to use this *Singleton* instance instead of creating an object of **RentalSystem**.
2. **[**9 Points**]** Currently, all vehicle, customer, and rental record details are lost upon exiting the program. To address this, implement the following methods in the **RentalSystem** class. Each method should write to the file in append mode to preserve existing entries.
   - saveVehicle(**Vehicle** *vehicle*): adds *vehicle* details to ***vehicles.txt***. Called inside addVehicle().
   - saveCustomer(**Customer** *customer*): adds *customer* details to ***customers.txt***. Called inside addCustomer().
   - saveRecord(**RentalRecord** *record*): adds rental record details to ***rental_records.txt***. Called at the end of rentVehicle() and returnVehicle() after a record is added to the rental history.
3. **[**9 Points**]** Add a private method called loadData() in the **RentalSystem** class to load previously saved data when the program starts. This method should:
   - Load vehicles from **vehicles.txt** and populate the vehicles list.
   - Load customers from **customers.txt** and populate the customers list.
   - Load rental records from **rental_records.txt** and populate the **rentalHistory** object.

   Call loadData() from the **RentalSystem** constructor to ensure all previously saved data is available upon launching the program. Parsing should match the format used in the corresponding saveVehicle, saveCustomer, and saveRecord methods.
4. **[**9 Points**]** Currently, the program allows adding multiple vehicles with the same license plate or customers with the same customer ID without any checks. Modify the addVehicle() and addCustomer() methods to verify whether the provided license plate or customer ID already exists before adding a new vehicle or customer. If a duplicate is found, prevent the addition and display an appropriate message. Change the return type of these methods from void to boolean, where *true* indicates a successful addition and *false* indicates duplication. You may use the provided findVehicleByPlate() and findCustomerById() if needed to support this check.

5. [9 Points] Refactor the constructor in the Vehicle class to improve code clarity and reduce duplication. Currently, both *make* and *model* are checked for *null* or empty values, and then formatted using repeated logic. Extract the formatting logic into a private helper method called *capitalize(String input)* that capitalizes the first letter and makes the rest lowercase. Then, call this helper method from the constructor when assigning values to *make* and *model*. This refactoring should preserve the original behavior while improving maintainability.

6. [5 Points] Perform the following:
   - Run the program.
   - Add two customers:
     - (**ID:** 001, **Name:** George)
     - (**ID:** 002, **Name:** Anne)
   - Add three vehicles:
     - (**Plate:** AAA111, **Make:** Toyota, **Model:** Corolla, **Year:** 2019)
     - (**Plate:** BBB222, **Make:** Honda, **Model:** Civic, **Year:** 2021)
     - (**Plate:** CCC333, **Make:** Ford, **Model:** Focus, **Year:** 2024)
   - Rent and return vehicles as follows:
     - Customer with ID 001 (George) rents AAA111 and CCC333.
     - Customerwith ID 002 (Anne) rents BBB222 and CCC333.
     - Customerwith ID 001 returns CCC333.
     - Customerwith ID 002 rents CCC333 again.
   - Display the available vehicles.
   - Show the full rental history.
   - Save the console output or take multiple screenshots of the output of the above steps.
   - Ensure you commit all code and data file changes.

Submission:

1. The final *.java* files, either edited or not, should be zipped in a zip file named *Task1_Code.zip*.
2. The console output taken as part of Subtask#6 (file name should be *Task1_Output.txt* or *Task1_Output.pdf*). No need to submit the .txt files used to save the data.
3. A screenshot showing at least **six Git commits**, one per each of the above subtasks (screenshot file name should be *Task1_Git.jpg*, *Task1_Git.png*, or *Task1_Git.pdf*).

Task 2: Testing [30 Points]

Use **JUnit 5** to create a test class called **VehicleRentalTest** and implement the test cases described below. Each test case should be placed in its own method annotated with @Test. You may use assertions such as assertTrue, assertFalse, assertEquals, assertNull, assertThrows, or fail as appropriate. If needed, include a setUp() method annotated with @BeforeEach to initialize shared objects used across multiple test methods.

1. [10 Points] **Vehicle License Plate Validation**

   Vehicle class:

- There is currently no method in the **Vehicle** class for validating license plate input. Add a private method isValidPlate(String plate) that returns *true* only if the plate is not *null*, not empty, and follows a valid format (e.g., three letters followed by three numbers).

- Update the setLicensePlate() method to call isValidPlate() before assigning the plate. If the plate is invalid, throw an *IllegalArgumentException* with an appropriate message.

RentalSystemApp class**:**

- In the section of the main method where the user enters a license plate, wrap the call to set the license plate in a try-catch block and display an error message if the plate is invalid.

VehicleRentalTest class**:**

- Create a test method testLicensePlateValidation() in which you will need to instantiate multiple **Vehicle** objects using the following data and use several assertions to validate them.
  - Test valid plates: *AAA100*, *ABC567*, and *ZZZ999*.
  - Test invalid plates: *empty* string, *null*, *AAA1000*, and ZZZ99.
  - Use assertTrue, assertFalse, and assertThrows as appropriate.

2. [10 Points**] Rent/Return Vehicle Validation**

VehicleRentalTest class**:**

- Create a test method testRentAndReturnVehicle() in which you will:
  - Instantiate a ***Vehicle*** and ***Customer*** objects.
  - Use a proper assertion to ensure that the vehicle is initially available.
  - Retrieve the single ***RentalSystem*** instance (here or in the setUp method)
  - Call the rentVehicle method for the created vehicle and customer objects, then use proper assertions to ensure that:
    - borrowing is successful (returns *true*)
    - The vehicle is marked as *RENTED*
    - Note that you will need to make rentVehicle return boolean to support this test case.
  - Try renting the same vehicle again and assert that it fails.
  - Call returnVehicle() for the same vehicle and customer objects, then use proper assertions to ensure that:
    - returning is successful.
    - The vehicle is marked as *AVAILABLE*
    - Note that you will need to make returnVehicle return boolean to support this test case.
  - Try returning the same vehicle again and assert that it fails.

3. [10 Points**] Singleton Validation**

VehicleRentalTest class**:**

- Create a test method testSingletonRentalSystem() to validate that the **RentalSystem** class enforces Singleton behavior, disallowing direct instantiation of the class. For this, you will use the Java reflection package, which allows accessing the metadata of Java code.
  - Import these classes: *java.lang.reflect.Constructor* and *java.lang.reflect.Modifier*.

- Use *Constructor<RentalSystem> constructor = RentalSystem.class.getDeclaredConstructor();* to return the constructor of the **RentalSystem** class. You will need to put your test code inside a *try-catch* block or add *throws Exception* as part of the test method's declaration.
- Use the *getModifiers* method to get the constructor's modifier.
- Then use a proper assertion to validate if the returned value equals *Modifier.PRIVATE*.
- Use RentalSystem.getInstance() to obtain an instance and assert it is not null.

Submission:

1. The complete JUnit test class file (*VehicleRentalTest.java*), containing all the test methods with their implementation. Please do not zip it with the other classes.
2. A screenshot showing test results in Eclipse (file name should be *Task2_Output.jpg*, *Task2_Output.png*, or *Task2_Output.pdf*).
3. A screenshot showing at least **three *Git* commits**, one for each of the above subtasks (screenshot file name should be *Task2_Git.jpg*, *Task2_Git.png*, or *Task2_Git.pdf*).

[OPTIONAL] Task 3: GUI [25 Bonus Points]

Develop a **JavaFX** GUI for the vehicle rental system, replacing the console menu. Create a new class called **RentalSystemGUI** that will handle all the necessary GUI components and their corresponding events. This class should include its own main() method, allowing it to run independently from the existing **RentalSystemApp** class.

1. The GUI should handle all the operations performed by the **RentalSystemApp** class, including adding vehicles, adding customers, renting, returning, displaying available vehicles, and showing full rental history. You can also include additional features if you want.
2. Users should input vehicle and customer details through text fields, select vehicles and customers from a list for renting and returning, and view available vehicles and rental history in a clear, user-friendly format.
3. You can use JavaFX components such as buttons, text fields, combo boxes, list views, and menus, ensuring proper error handling and an intuitive design**.**

Submission:

1. Submit the *RentalSystemGUI.java* file. Please do not zip it with the other classes.
2. The GUI must be developed using JavaFX. If Swing, AWT, or Applet is used, then no points will be awarded.
3. The GUI must run without errors to qualify for bonus points; otherwise, no points will be awarded.
4. You are also expected to use *Git* versioning when working on this task, but you are not required to submit any screenshots showing your commits.