TRENT UNIVERSITY

COIS 2240H – Software Design & Modelling

Assignment 2 — UML Design and Modelling

Total Points: 100 – Due March 16, 2025

Note: The assignment should be completed through individual efforts (i.e., group submissions are not accepted). Any issues related to academic integrity, such as plagiarism, copying from another student, 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 for late submission, and no more than 5 days late. If there is a late second attempt, the submission will be considered late.

Objectives:

Design and model software using UML diagrams based on given system requirements, apply object-oriented principles in Java, and generate Java code from the software design.

Introduction:

For this assignment, you will demonstrate your understanding of software design and modeling using UML diagrams and their implementation in Java. Specifically, you are expected to:

1. Create UML diagrams to model a software system's relationships, interactions, and behavior and between different classes and objects.
2. Generate Java source code that reflects the design of the software.
3. Clean the generated source code to ensure completeness and relevance.

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

Task 1 [60 Points]

You have already initiated a vehicle rental system in Assignment#1. Now, we will enhance the system by adding more classes. You are not required to write the code for these classes at this stage.

1. **Truck Class** (inherits from **Vehicle**)**:**
   - **Additional attributes** (private):
     - cargoCapacity (double): Number of seats in the car (greater than zero)
   - **Constructor** (public):
     - To initialize all attributes and pass values of the inherited attributes to the superclass
   - **Method** (public):
     - Override getInfo() to get the details of the parent class and extend them to include the cargo capacity
2. **SportCar Class** (inherits from **Car**, and is declared final to prevent further inheritance)**:**
   - **Additional attributes** (private):
     - horsepower (int): Engine power in horsepower unit
     - hasTurbo (boolean): Whether the car has a turbocharged engine
   - **Constructor** (public):

        - To initialize all attributes and pass values of the inherited attributes to the superclass
    - **Method** (public):
        - Override getInfo() to get the details of the parent class and extend them to include the *horsepower* and *hasTurbo* details

3. **Customer Class:**
    - **Attributes** (private)**:**
        - customerId (String): Unique identifier for each customer
        - name (String): Customer's full name
        - phoneNumber (String): Customer's phone number
    - **Constructor** (public):
        - To initialize all customer attributes
    - **Methods** (public)**:**
        - toString(): To return a formatted string with all customer details
4. **RentalRecord Class** (represents a single rental transaction with the below details**):**
    - **Attributes** (private)**:**
        - vehicle (Vehicle): The vehicle involved in the transaction
        - customer (Customer): The Customer who rents/returns the vehicle
        - recordDate (LocalDate): When the vehicle was rented or returned
        - totalAmount (double): Total rental amount for this transaction
        - recordType (String): Type of transaction record: "*RENT*" or "*RETURN*"
    - **Constructor** (public):
        - To initialize all rental record attributes
    - **Methods** (public)**:**
        - toString(): returns a string representation of the rental record, which includes the car's license plate, customer name, rental dates, and the total amount
5. **RentalHistory Class:**
    - **Attributes** (private)**:**
        - rentalRecords: an *ArrayList* of RentalRecord objects
    - **Methods** (public)**:**
        - addRecord (RentalRecord): Adds a new rental record to the rental history
        - getRentalHistory(): Returns the entire rental history
        - getRentalRecordsByCustomer(String): Retrieves all rental records for a specific customer name
        - getRentalRecordsByVehicle(String): Retrieves all rental records for a specific vehicle license plate
6. **Changes** to **RentalSystem Class**:
    - **Attributes** (private)**:**
        - Remove the rentalRecords variable that was of type ***Map***
        - Add an object of RentalHistory object
    - **Methods** (public)**:**

- Update the rentVehicle method to have the parameters (Vehicle, Customer, LocalDate, double), where LocalDate is for the rental date and double is for the rental amount. This method should perform the regular checking as before. It will create a new rental record and add it to the rental history and specify the transaction type as "*RENT*"
- Update the returnVehicle method to have the parameters (Vehicle, Customer, LocalDate, double), where LocalDate is for the return date and double is for extra fees to be paid upon return, if any. This method should perform the regular checking as before. It will create a new rental record and add it to the rental history and specify the transaction type as "RETURN"
- Other methods, rentVehicle(Vehicle) and displayAvailableVehicles(), should remain unchanged

Requirements:

1. **[**20 Points**]** Draw a **Class Diagram** to represent the entire structure of the vehicle rental system (consisting of the original and new classes), properly labelled to show class names, attributes, methods, and different types of associations. You must use UmpleOnline for this purpose.
2. **[**20 Points**] Generate the Java source code** using **Umple**. The code must reflect the new software system you have modelled. Make sure the code is consistent with the original source code implemented in ***Assignment#1***. Make sure your code is clean and simple (follow the guidelines in Point#3 from "***Umple Uses***" in ***Lab #4*** to produce less complex, maintainable code). You are **NOT** required to implement the functionality of the new methods introduced in this assignment; an empty method body (with a proper return) is sufficient. The implementation of all methods will be part of Assignment#3.
3. **[**20 Points**]** Draw a **Sequence Diagram** to illustrate the interactions of the vehicle rental process, starting with the user adding a vehicle to the rental system through the rental app, specifying type of vehicle, then renting the vehicle to a customer, and handling any checking, updating vehicle status, rental record, and history. The same thing also applies when the user returns a vehicle. After that, the user can display available vehicles, show rental history, then exit the application. You can use any available tool (e.g., *Visual Paradigm, ArgoUML, Word, PowerPoint, Visio*, or *others*), since Umple does not support this type of UML diagrams.

Submission:

1. A **screenshot** of the designed Class Diagram. Drawing by hand is discouraged. Notes:
   - Include your name at both the top and bottom of the class diagram. If no name is displayed, you will lose all marks for this task. *You will lose marks for missing key details in your design, including attributes, methods, relationships, or associations.*
2. A **screenshot** of the designed Sequence Diagram. Drawing by hand is discouraged. Notes:
   - Include your name at the top and bottom of the sequence diagram. If no name is displayed, you will lose all marks for this task. *You will lose marks for missing key details in your design, including actors, objects, messages, lifelines, conditions, or loops.*
3. One **Java file** named **Task1.java** contains all the classes for the system. Notes:
   - Include your name as a comment at the top and bottom of the Java code. If no name is provided, you will lose all marks for this task. You will lose marks if your code does not reflect the class diagrams or if your code is too complex.

Task 2 [40 Points]

Requirements:

1. **[**20 Points**]** Draw a **State Diagram** that captures the different states of a vehicle.
   - A vehicle can always start in the `Available` state. After a vehicle is rented, it transitions to `Rented`. When returned, a rented vehicle moves back to `Available`. When a vehicle is reserved by a customer, it moves to `Reserved`. After the customer picks up a reserved vehicle, it should transition from `Reserved` to `Rented`. If the reservation is canceled, it returns to `Available`. If the reservation is for more than a week, it is considered `Available` again. If the vehicle requires maintenance, it can move from any state to `Maintenance`. If a vehicle finishes its repairs, it can return to `Available`. If the vehicle is deemed unfit for rental at any state, it transitions to `OutOfService`.
   - Ensure your diagram follows the State Diagram guidelines described in lectures and is labelled properly to clearly show the states and transitions as well as the actions that trigger them.
2. **[**20 Points**]** Draw an **Activity Diagram** that captures the actions and decisions for renting a vehicle.
   - The system receives a customer's request to rent a vehicle and performs checks concurrently. It first verifies if the vehicle is available; if it's rented, under maintenance, or out of service, the process terminates. If the vehicle is reserved, the system checks if the reservation is for more than a week, in which case it considers the vehicle available. At the same time, the system checks the customer's eligibility by checking their age and driver's license. If the vehicle is available and the customer is eligible, the system proceeds with the rental.
   - Ensure your diagram follows the activity diagram guidelines described in lectures and is labelled properly to clearly show the concurrent operations/conditions of the rental process.
   - ***Bonus 5 Points:*** if you properly equip the activity diagram with swimlanes.

Submission:

1. A **screenshot** of the designed State Diagram. Drawing by hand is discouraged. Notes:
   - Include your name at the top and bottom of the state diagram. If no name is displayed, you will lose all marks for this task.
   - You will lose marks for missing key details in your design, such as states (including a start state and end states), transitions (including any elapsed-time or conditional transitions), and proper labels.
2. A **screenshot** of the designed Activity Diagram. Drawing by hand is discouraged. Notes:
   - Include your name at the top and bottom of the activity diagram. If no name is displayed, you will lose all marks for this task.
   - *You will lose marks for missing key details in your design, such as forks, joins, rendezvous splits, decision nodes, merge nodes, and proper labels.*

General Submission Note: Please submit your screenshots as individual image files or combine them into a single PDF file for each task. Avoid submitting any zip files.