CA Project

[**Object Oriented Programming I**]

<name, student number>
TUS Athlone, **MSc** Software Design with AI , AL_KSAIM_9, Semester 1
<date>

Contents

1. Introduction

The Vehicle Rental Management System is a Java-based desktop application designed for management of a small vehicle rental business. This application allows administrators to manage renting of bikes and scooters, via an intuitive GUI built using Java Swing. The application showcases core object-oriented programming (OOP) concepts as well as more modern Java features.

Main Features:

- **Vehicle Inventory Management:** Users can add new vehicles, edit existing ones, and delete outdated or retired vehicles.
- **Rental Process and State Tracking:** The system keeps track of each vehicle's status, as AVAILABLE, ON_RENT, or SERVICED. Vehicles can be rented out with specified duration.
- **Error Handling:** Incorporates error handling with custom exceptions to manage invalid operations and ensure system stability.
- **Java Features:** Utilizes Java features such as sealed classes, records, lambda expressions, and switch expressions for cleaner and more efficient code.
- **Graphical User Interface:** Basic GUI build using Java Swing framework to enhance user experience. (Fig.1)

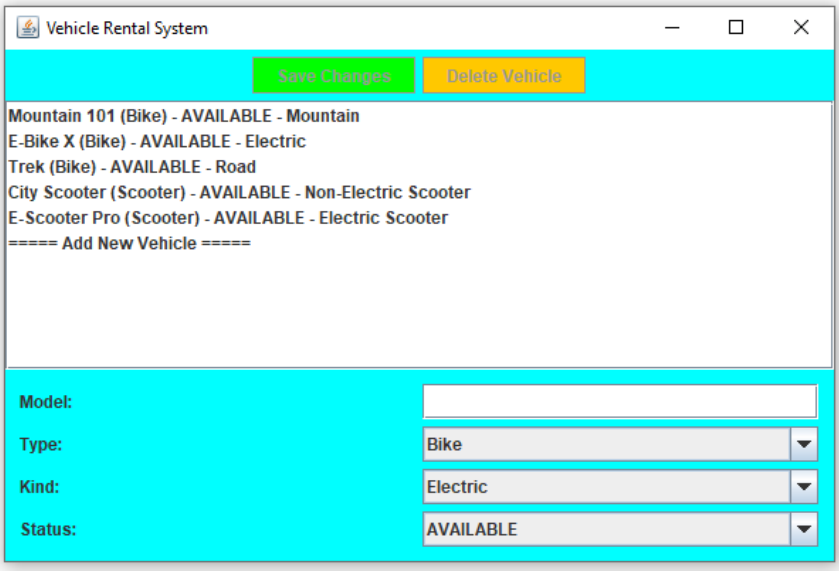

Figure 1: Graphical User Interface

1.1. User Stories

1.1.1. Functional User Stories

Inventory Management

User Story F1.1: As an administrator, I want to add new vehicles to the inventory through the GUI, so I can track all available bikes and scooters.

- **Acceptance Criteria:**
    - I can enter the vehicle model and select its type (Bike or Scooter) and specific kind (e.g., Electric Bike, Mountain Bike, Electric Scooter).
    - The newly added vehicle is marked as AVAILABLE by default.
    - I receive a confirmation message once the vehicle is added successfully.

User Story F1.2: As an administrator, I want to edit vehicle details using the GUI, so I can update incorrect or outdated information.

- **Acceptance Criteria:**
    - I can select a vehicle from the list displayed in the GUI.
    - I can update the vehicle model, subtype/kind, and status (AVAILABLE, ON_RENT, SERVICED).
    - I cannot update type e.g. change bicycle to a scooter, as this can only be a result of user mistake.
    - Changes are saved upon confirmation, and a success message is displayed.

User Story F1.3: As an administrator, I want to delete vehicles from the inventory using the GUI, so I can remove outdated or retired bikes and scooters.

- **Acceptance Criteria:**
    - I can select a vehicle from the list in the GUI to delete.

- I am prompted to confirm the deletion.
- Upon confirmation, the vehicle is removed from the inventory and no longer appears in the list.
- Vehicles that are currently rented cannot be deleted.

Rental Process

User Story F2.1: As an administrator, I want to rent a vehicle to a customer through the GUI, so they can use it for a specific duration.

- **Acceptance Criteria:**
  - I can see a list of all available vehicles in the GUI.
  - I can select a vehicle to rent.
  - I can specify the rental duration in hours via an input dialog.
  - The vehicle status updates to ON_RENT, and the expected return time is displayed.

User Story F2.2: As an administrator, I want to return a vehicle to the system using the GUI, so I can complete the rental process.

- **Acceptance Criteria:**
  - I can select the rented vehicle from the list.
  - The vehicle's status is updated to SERVICED or AVAILABLE.
  - A confirmation message indicates the successful return of the vehicle.

System Functionality and Reporting

User Story F3.1: As an administrator, I want to view all vehicles in the inventory through the GUI, so I can track their details and availability.

- **Acceptance Criteria:**
  - The GUI displays a list of all vehicles with details such as model, type, kind, and status.
  - Vehicles show their status (AVAILABLE, ON_RENT, SERVICED) and return time if they are rented.

1.1.2. Technical User Stories

User Story T1.1*:* As a developer, I want to implement a comprehensive class hierarchy with modern Java features.

- **Acceptance Criteria:**
    - Creation of a sealed abstract Vehicle class that encapsulates common attributes and methods.
    - Bike and Scooter classes extend Vehicle, demonstrating inheritance.
    - Constructors utilize this() and super() for initialization.
    - Overridden methods in subclasses demonstrate polymorphism, including the use of super to invoke superclass methods.
    - Encapsulation is maintained with private fields and public getter/setter methods.
    - Method overloading and varargs are used in meaningfully throughout the source code.

User Story T1.2: As a developer, I want to manage vehicle status and type effectively using enums and modern control flow statements.

- **Acceptance Criteria:**
    - Use of enums for vehicle types (BikeType) and statuses (BikeStatus).
    - Implementation of switch expressions and pattern matching for type and status logic.
    - Defensive copying is used to safely handle mutable objects like LocalDateTime.
    - Sealed classes are used to restrict subclassing of Vehicle.

User Story T1.3: As a developer, I want to demonstrate advanced interface functionality.

- **Acceptance Criteria:**

- The Rentable interface includes default and static methods, as well as a private helper method.
- Vehicle or its subclasses implement the Rentable interface.
- Default methods provide shared functionality, while private methods support internal logic.
- Static methods offer utility functions applicable to all rentable items.

User Story T1.4: As a developer, I want to implement robust error handling with custom exceptions.

- **Acceptance Criteria:**
  - Custom exceptions like VehicleCurrentlyRentedException are created for specific error scenarios.
  - Both checked and unchecked exceptions are used appropriately.
  - Try-catch blocks are implemented to handle exceptions in the GUI.
  - Input validation uses lambdas and predicates where suitable.

User Story T1.5: As a developer, I want to implement efficient and flexible data storage and manipulation.

- **Acceptance Criteria:**
  - Use of ArrayList and List for dynamic vehicle inventory management.
  - Demonstration of method references and lambdas for streamlined operations.
  - Varargs are used in methods like addVehicles() for flexibility.
  - StringBuilder is utilized for efficient string concatenation in methods such as toString().

User Story T1.6: As a developer, I want to manage vehicle return times and timestamps effectively.

- **Acceptance Criteria:**
  - Use of the java.time API for date and time management.
  - Return times are formatted using DateTimeFormatter.

- Rental durations and return timestamps are accurately calculated and stored.
- Defensive copying is used when handling date objects to prevent unintended modifications.

User Story T1.7: As a developer, I want to demonstrate immutability and the use of records.

- **Acceptance Criteria:**
  - Creation of custom record types like RentalTransaction for immutable data storage.
  - Records are used where appropriate to reduce boilerplate code.
  - Immutable classes are used to represent data that should not change after creation.

User Story T1.8: As a developer, I want to implement modern Java features for cleaner and more efficient code.

- **Acceptance Criteria:**
  - Sealed classes are used to restrict subclassing and maintain control over the class hierarchy.
  - Lambdas and method references are integrated for event handling and data processing.
  - Local variable type inference (var) is used where it enhances readability.
  - Switch expressions and pattern matching are employed for concise and clear logic handling.

2. Implementation

2.1. Object-Oriented Design

2.1.1. Inheritance and Polymorphism

The Vehicle class serves as a sealed abstract base class for specific vehicle types (Bike and Scooter) (See Appendix 1: Class UML Diagram) These subclasses inherit common attributes (model, type, status) and behaviors (rent, returnItem) from Vehicle. Polymorphism is demonstrated through method overriding, particularly in the toString() method where each subclass extends the parent's implementation to add type-specific information(Fig.2).

```
public sealed abstract class Vehicle implements Rentable permits Bike, Scooter {
    private final String type;
    private String model;
    private VehicleStatus status;
    // Constructor and common methods...
    public abstract String getSpecificType();
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(model).append(" (").append(type).append(") - ");
        sb.append(status);
        // ... rental time formatting ...
        return sb.toString();
    }
}
public final class Bike extends Vehicle {
    private BikeType bikeType;

    public Bike(String model, BikeType bikeType) {
        super(model, "Bike");
        this.bikeType = bikeType;
    }
    @Override
    public String getSpecificType() {
        return bikeType.getDisplayName();
    }
    @Override
    public String toString() {
        return super.toString() + " - " + getSpecificType(); // Polymorphism example
    }
}
```

Figure 2: Example of Inheritance and Polymorphism

2.1.2. Encapsulation

Classes such as Bike and Scooter use private fields with public getter and setter methods to control access to their data. This is to prevent external classes from directly modifying internal states, ensuring data integrity.

```
public final class Bike extends Vehicle {
    private BikeType bikeType; // Private field

    public BikeType getBikeType() { // Public getter
        return bikeType;
    }

    public void setBikeType(BikeType bikeType) { // Public setter
        this.bikeType = bikeType;
    }

    // Other methods...
}
```

Figure 3: Encapsulation in Bike class

2.1.3. Interfaces with Default and Static Methods

The Rentable interface defines the contract for rentable items with various types of interface methods(Fig.4)

- **Abstract Methods:**
    - rent(int hours): Must be implemented to handle rental logic
    - returnItem(boolean needsService): Must be implemented to handle return logic
    - isUnderMaintenance(): Must be implemented to check maintenance status

- **Default Method** inspectBeforeRent() provides a default implementation that can be used or overridden by implementing classes, promoting code reuse.
- **Private Interface Method** performGeneralInspection() is encapsulated within the interface, keeping the API clean and focused.

- **Static Method** sanitize() is a utility function relevant to the interface that can be called without an instance.

```
public interface Rentable {
    static void sanitize() { // Static method
        System.out.println("Sanitizing all rentable items.");
    }

    void rent(int hours) throws VehicleCurrentlyRentedException;

    void returnItem(boolean needsService);

    default void inspectBeforeRent() { // Default method
        if (isUnderMaintenance()) {
            throw new IllegalStateException("Cannot rent item under
maintenance.");
        }
        performGeneralInspection();
    }

    private void performGeneralInspection() { // Private method
        System.out.println("Performing general inspection.");
    }

    boolean isUnderMaintenance();
}
```

Figure 4: Rentable interface with default and static methods

2.2. Core Java Features

2.2.1. Constructors and this() Keyword

Constructors in Bike and Scooter classes demonstrate constructor chaining using this() keyword and parent constructor calling using super() keyword. These techniques provide flexible object instantiation while maintaining code reuse (Fig.5)

```
public Scooter(String model) {
   this(model, false); // Chains to the full constructor with default value
}

public Bike(String model, BikeType bikeType) {
   super(model, "Bike");  // Calls Vehicle constructor
   this.bikeType = bikeType;
}
```

Figure 5: Constructors using this() and super() keyword

2.2.2. Enums

The application uses enums to define fixed sets of constants(Fig.6)

1. VehicleStatus enum defines possible vehicle states
2. BikeType enum demonstrates more advanced enum features. These provide type safety, prevent invalid values, and add functionality like display name formatting and string-to-enum conversion with validation.

```
public enum VehicleStatus {
   AVAILABLE,
   ON_RENT,
   SERVICED
}

public enum BikeType {
   ELECTRIC("Electric"),
   MOUNTAIN("Mountain"),
   ROAD("Road"),
   CHILD("Child");

   private final String displayName;

   BikeType(String displayName) {
      this.displayName = displayName;
   }

   public static BikeType fromDisplayName(String displayName) {
      // Conversion logic from display name to enum
      // with validation
   }
}
```

Figure 6: Enums for VehicleStatus and BikeType

2.2.3. Exception Handling

The system distinguishes between checked and unchecked exceptions to handle error conditions appropriately, enhancing reliability.

- **Checked Exception** VehicleCurrentlyRentedException is an example of custom exception used for anticipated issues that callers can recover from. This enforces handling through the compiler(Fig.7).
- **Unchecked Exceptions**: Indicate programming errors or unexpected conditions, not requiring explicit handling(Fig.8).

```
public class VehicleCurrentlyRentedException extends Exception {
    public VehicleCurrentlyRentedException(String message) {
        super(message);
    }
}
// Usage in VehicleRentalSystem
public void removeVehicle(Vehicle vehicle) throws VehicleCurrentlyRentedException {
    if (vehicle.getStatus() == VehicleStatus.ON_RENT) {
        throw new VehicleCurrentlyRentedException("Cannot delete a vehicle that is currently rented.");
    }
    vehicles.remove(vehicle);
}

// Usage in Vehicle class
public void rent(int hours) throws VehicleCurrentlyRentedException {
    inspectBeforeRent();
    if (status != VehicleStatus.AVAILABLE) {
        throw new VehicleCurrentlyRentedException("Vehicle is not available for rent.");
    }
    status = VehicleStatus.ON_RENT;
    rentalEndTime = LocalDateTime.now().plusHours(hours);
}
```

Figure 7: Custom Exception and its usage

```
default void inspectBeforeRent() {
    if (isUnderMaintenance()) {
        throw new IllegalStateException("Cannot rent item under maintenance.");
    }
    performGeneralInspection();
}
```

Figure 8: Unchecked exception in Rentable interface

2.3. Advanced Java Features

2.3.1. Lambdas and Method References

The application uses lambda expressions and method references in multiple areas of the codebase. This use of functional programming features simplifies code and improves readability, especially in event handling and collection processing operations(Fig. 9,10 and 11).

```
vehicleListModel = new DefaultListModel<>();
    system.getVehicles().forEach(vehicleListModel::addElement); // Method reference
```

Figure 9: Method References in VehicleRentalGUI for List Population

```
// In VehicleRentalGUI constructor - simple event listeners
saveButton.addActionListener(e -> onSaveChanges());
    deleteButton.addActionListener(e -> onDeleteVehicle());
    typeComboBox.addActionListener(e -> updateVehicleTypeOptions());
```

Figure 10: Lambda Expressions for Event Handling in VehicleRentalGUI

```
// In VehicleRentalSystem class
public List<Vehicle> filterVehicles(Predicate<Vehicle> condition) {
  return vehicles.stream()
      .filter(condition)
      .collect(Collectors.toList());
}

// Example usage with method reference
public List<Vehicle> getAvailableVehicles() {
  return filterVehicles(Vehicle::isAvailable);
}

// Example usage with lambda expression to filter by specific status
public List<Vehicle> getRentedVehicles() {
  return filterVehicles(vehicle -> vehicle.getStatus() == VehicleStatus.ON_RENT);
}
```

Figure 11: Predicates with Lambda Expressions for Vehicle Filtering

2.3.2. Records and Immutability

The RentalTransaction record provides an immutable data structure for rental information. This implementation ensures data integrity as all fields are final and automatically generates equals(), hashCode(), and toString() methods. The record's concise syntax improves code readability

```
public record RentalTransaction(
        Vehicle vehicle,
        LocalDateTime rentalTime,
        int durationHours) {
ι
```

Figure 12: RentalTransaction record

2.3.3. Sealed Classes and Pattern Matching

The application uses sealed class to restrict inheritance and pattern matching(Fig.13) for type-safe operations. This approach provides compile-time safety for class hierarchy. Pattern matching (using 'instanceof') allows to check and convert object types in one step, making the code cleaner and less error-prone compared to older approaches .

```
// Sealed class definition
public sealed abstract class Vehicle implements Rentable
        permits Bike, Scooter {
    // Implementation
}

// Pattern matching in VehicleRentalGUI
if (selected instanceof Vehicle vehicle) {
        modelField.setText(vehicle.getModel());

        if (vehicle instanceof Bike bike) {
        vehicleTypeComboBox.setSelectedItem(bike.getBikeType().getDisplayName());
        } else if (vehicle instanceof Scooter scooter) {
        vehicleTypeComboBox.setSelectedItem(scooter.isElectric() ?
        "Electric Scooter" : "Non-Electric Scooter");
        }
    }
```

Figure 13: Sealed class definition and pattern matching

2.3.4. Switch Expressions

Modern switch expressions in the GUI provide clear, compact handling of different vehicle status scenarios. For example, when saving vehicle changes.

```
switch (newStatus) {
   case ON_RENT -> {
      try {
         String hoursStr = JOptionPane.showInputDialog(this, "Enter rental hours:", "Rent Vehicle",
JOptionPane.PLAIN_MESSAGE);
         if (hoursStr == null) return;  // User cancelled

         int hours = Integer.parseInt(hoursStr);
         if (hours <= 0) {
            JOptionPane.showMessageDialog(this, "Please enter a positive number of hours.");
            return;
         }
         vehicle.rent(hours);
         // Show success message with return time
      } catch (Exception ex) {
         // Handle errors
      }
   }
   case AVAILABLE, SERVICED -> {
      boolean needsService = newStatus == VehicleStatus.SERVICED;
      vehicle.returnItem(needsService);
      // Show success message
   }
}
```

Figure 14: Switch expression handling different statuses.

2.3.5. Defensive Copying

Technique of defensive copying in used in VehicleRentalSystem class. By returning a new list, we ensure that callers cannot modify the internal vehicles list. This prevents external code from adding, removing, or altering the vehicles directly.

```
public class VehicleRentalSystem {
   private final List<Vehicle> vehicles;

   public List<Vehicle> getVehicles() {
      return new ArrayList<>(vehicles);  // Returns a copy, not the original
   }
}
```

Figure 15: Example of defensive copying in VehicleRentalSystem class

2.4. Java Core API and Collections

2.4.1. Collections and Data Management

The VehicleRentalSystem class demonstrates several features of Java collections. This implementation showcases modern Java collections features including:

- Type safety through generics.
- Flexible collection interfaces
- Varargs for convenient method calls
- Stream API for data processing
- Business logic integration with collections operations

```
// Core Collection Usage
public class VehicleRentalSystem {
   private final List<Vehicle> vehicles;    // Using interface type for flexibility

   public VehicleRentalSystem() {
      vehicles = new ArrayList<>();
   }
}

// Single vehicle addition
public void addVehicle(Vehicle vehicle) {
   vehicles.add(vehicle);
}

// Multiple vehicles using varargs
public void addVehicles(Vehicle... newVehicles) {
   vehicles.addAll(Arrays.asList(newVehicles));
}

// Stream operations for filtering
public List<Vehicle> filterVehicles(Predicate<Vehicle> condition) {
   return vehicles.stream()
         .filter(condition)
         .collect(Collectors.toList());
}

// Safe removal with business logic
public void removeVehicle(Vehicle vehicle) throws VehicleCurrentlyRentedException {
   if (vehicle.getStatus() == VehicleStatus.ON_RENT) {
      throw new VehicleCurrentlyRentedException("Cannot delete a vehicle that is currently rented.");
   }
   vehicles.remove(vehicle);
}
```

Figure 16 : Use of collections in VehicleRentalSystem

2.4.2. Strings and StringBuilder

StringBuilder in this application is used in toString() methods. StringBuilder is mutable and modifies the same object with each append(), while String concatenation creates new String objects each time. This makes StringBuilder more memory-efficient, especially when building strings in loops or with multiple concatenations.

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(model)
            .append(" (")
            .append(type)
            .append(") - ")
            .append(status);

    if (status == VehicleStatus.ON_RENT && rentalEndTime != null) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
        sb.append(" - Return by: ")
                .append(rentalEndTime.format(formatter));
    }
    return sb.toString();
}
```

Figure 17: toString() method using StringBuilder

2.4.3. Date and Time API

The LocalDateTime class provides clean date/time calculations, while DateTimeFormatter enables user-friendly date/time display. This application uses modern Java Date and Time API in several ways throughout the source code(Fig.).

```
// Managing rental duration
public void rent(int hours) throws VehicleCurrentlyRentedException {
   // Validation and status checks...
   status = VehicleStatus.ON_RENT;
   rentalEndTime = LocalDateTime.now().plusHours(hours);  // Calculate return time
}
// Formatting for display in toString() method
if (status == VehicleStatus.ON_RENT && rentalEndTime != null) {
   DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
   sb.append(" - Return by: ")
         .append(rentalEndTime.format(formatter));
}
// Generating success message in GUI
var formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
JOptionPane.showMessageDialog(
      this,
      "Vehicle rented successfully until " +
            vehicle.getRentalEndTime().format(formatter),
      "Rental Success",
      JOptionPane.INFORMATION_MESSAGE
);
```

Figure 18: Examples of LocalDateTime and DateTimeFormatter usage throughout the application

3. Conclusion

3.1. Learning Outcomes

The goal of this project was to use a robust set of Java and OOP features on example of simple java application. Perhaps the biggest challenge was to find meaningful use for all required java features in such a small application. At the end, functional and technical user stories described in Section 1 were successfully implemented. Using these Java features provided valuable insights into modern programming practices. Understanding how each feature enhances the application reinforced the importance of thoughtful design and utilizing capabilities of Java.

3.2. Future Enhancements

Real world usefulness of this application would be improved by incorporating following features.

- Persistent Storage: Integration of text file or a database to store vehicle and rental data persistently, so that data persist beyond application runtime.
- Logging Customer Data: Storing information about which customer is renting which vehicle for how long.
- User Authentication: Implementing user roles and authentication mechanisms.
- Enhancing GUI Features: Introducing more interactive elements, such as search filters and detailed vehicle views, to improve user experience.

Appendix 1: Class UML Diagram