DESIGN AND IMPLEMENTATION OF A MOTORCYCLE SERVICE MANAGEMENT APPLICATION USING OBJECT-ORIENTED PROGRAMMING

By:

Group 4

Richie Corneliu	(24091397055)
Raza Agha Habibillah	(24091397059)
Jihan Salma Salsabila	(24091397068)
Diandra Dani Dhaviza	(24091397070)
Iqbal Pahargja Dinata	(24091397133)

INFORMATICS MANAGEMENT STUDY PROGRAM

FACULTY OF VOCATIONAL

SURABAYA STATE UNIVERSITY

2025

TABLE OF CONTENTS

List Of Figure

CHAPTER I
INTRODUCTION

1.1 Background

The rapid development of information technology has encouraged the utilization of software in various aspects of daily life, including data management and routine activities. One area that requires a well-structured data management system is vehicle management, particularly in recording vehicle data and maintenance or service history.

In general, vehicle service records are still maintained manually or rely solely on the vehicle owner's memory. This condition can lead to various problems, such as poorly documented service data, forgotten service schedules, and difficulties in tracking complete maintenance histories. Therefore, an application is needed to assist users in managing vehicle data and service history in a systematic and organized manner.

In software development, the Object-Oriented Programming (OOP) approach is one of the most widely used paradigms because it can represent real-world problems in the form of objects. By applying OOP concepts such as encapsulation, inheritance, and polymorphism, applications can be developed in a more structured, maintainable, and scalable manner.

Based on these issues, the author developed an Object-Oriented Programming (OOP)-based application named MotoCare to support the management of motorcycle data and service history. This application was developed using the Python programming language, implementing OOP concepts and equipped with a graphical user interface (GUI) to ensure ease of use for users.

1.2 Problem Formulation

The problem formulations in this study are as follows:

1. How can an application be designed to manage motorcycle data and service history efficiently using an Object-Oriented Programming approach?
2. How can Object-Oriented Programming concepts be applied to model real-world entities such as motorcycles, users, and service records within an application?

3. How can a graphical user interface (GUI) be implemented to facilitate user interaction in managing motorcycle and service data?
4. How can the developed application support structured data management while maintaining code modularity and maintainability?

1.3 Research Objectives

The objectives of developing the MotoCare application are as follows:

1. To design and implement a motorcycle data and service history management application based on Object-Oriented Programming.
2. To apply Object-Oriented Programming concepts in practical software development, including encapsulation, inheritance, and polymorphism.
3. To develop an application with a simple and user-friendly graphical user interface (GUI).
4. To fulfill the requirements of the Final Examination (UAS) for the Object-Oriented Programming course.

CHAPTER II
CLASS DIAGRAM AND OBJECT RELATIONS

2.1 Diagram Class

FIGURE 2.1 DIAGRAM CLASS

The class diagram illustrates the structure of the MotoCare application and the relationships between its main components. The system is designed using Object-Oriented Programming principles and consists of five main classes: Vehicle, Motorcycle, ServiceRecord, User, and MotoCareApp.

2.2 Explanation

- Vehicle Class

  The Vehicle class is an abstract base class that represents a general vehicle. It contains common attributes such as license plate number, brand, model, and production year. The class defines an abstract method get_info(), which must be implemented by all

subclasses. This design ensures that every specific type of vehicle provides its own implementation for displaying vehicle information, while sharing the same basic structure.

- Motorcycle Class

  The Motorcycle class is a subclass of Vehicle and represents a specific type of vehicle managed by the application. In addition to inheriting all attributes from the Vehicle class, it introduces new attributes such as current_km to store the latest mileage and service_records to store the service history.

  The class provides methods to add a new service record, retrieve the most recent service, and override the get_info() method. This demonstrates the use of inheritance and polymorphism, allowing the motorcycle to extend and customize the behavior defined in the base class.

- ServiceRecord Class

  The ServiceRecord class represents a single service activity performed on a motorcycle. It stores information such as service date, mileage at the time of service, service type, cost, and additional notes.

  This class is associated with the Motorcycle class in a one-to-many relationship, meaning that one motorcycle can have multiple service records. The __str__() method is implemented to provide a readable string representation of a service record, which is useful for displaying service history in the user interface.

- User Class

  The User class represents the owner of the motorcycles. It contains the user's name and a collection of motorcycles owned by the user. The class provides methods to add a motorcycle, retrieve a motorcycle based on its license plate number, and access the list of motorcycles.

  The relationship between User and Motorcycle is one-to-many, as a single user can own multiple motorcycles.

- MotoCareApp Class

  The MotoCareApp class acts as the application controller and serves as an intermediary between the user interface and the underlying data models. It holds a reference to a User object and provides high-level operations such as adding a motorcycle, adding or updating service records, retrieving service history, and removing motorcycles.

  By centralizing application logic in this class, the system maintains a clear separation between the user interface and the data model.

2.3 Relationships Overview

The class diagram demonstrates several key relationships:

- **Inheritance** is applied between the Vehicle and Motorcycle classes, allowing shared attributes and behavior to be reused.
- **Association** is used to model the relationships between User and Motorcycle, as well as between Motorcycle and ServiceRecord.
- **Multiplicity** indicates that one user can own multiple motorcycles, and one motorcycle can have multiple service records.

These relationships help ensure that the system structure is logical, modular, and easy to maintain.

CHAPTER III
IMPLEMENTATION AND APPLICATION FLOW

3.1 Application Structure

The MotoCare application is implemented using the Python programming language and follows a modular structure to separate responsibilities between data models, application logic, and the user interface.

The project is organized into several files and folders as shown below:

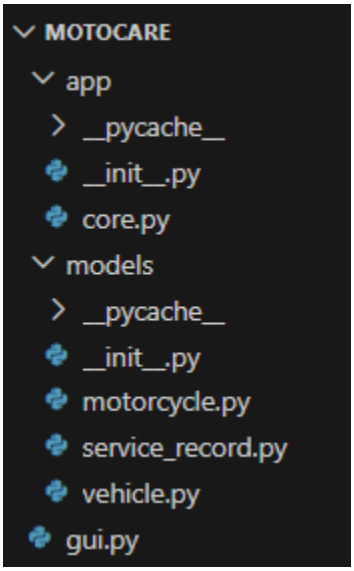

FIGURE 3.1 APPLICATION STRUCTURE

The models directory contains classes that represent the core data entities of the system. The app directory contains the main application logic, while the graphical user interface is implemented in gui.py. This structure improves code readability and maintainability.

3.2 Implementation of Core Components

3.2.1 Model Layer

The model layer consists of the Vehicle, Motorcycle, and ServiceRecord classes. These classes define the structure of the data and encapsulate the business rules related to vehicles and service history.

- The Vehicle class provides a general abstraction for vehicles.
- The Motorcycle class extends Vehicle and manages service records and mileage updates.
- The ServiceRecord class stores detailed information for each service activity.

All data manipulation related to motorcycles and service records is handled within this layer, ensuring that the data remains consistent and well-structured.

3.2.2 Application Logic Layer

The application logic is implemented in core.py, which contains the User and MotoCareApp classes.

The User class manages the ownership of motorcycles, while the MotoCareApp class acts as a controller that connects the user interface with the data models. Operations such as adding motorcycles, deleting motorcycles, adding service records, updating service records, and retrieving service history are performed through methods provided by MotoCareApp.

This approach prevents the user interface from directly accessing or modifying the data structures, thereby improving encapsulation and reducing coupling between components.

3.2.3 User Interface Layer

The graphical user interface is developed using the Tkinter library. All GUI-related components are encapsulated within the MotoCareGUI class in gui.py.

The interface is divided into three main sections:

- **Motor Input Section**
  Allows the user to input motorcycle data such as license plate, brand, model, production year, and current mileage
- **Motorcycle List Section**
  Displays the list of motorcycles owned by the user. The user can select a motorcycle or remove it from the list.
- **Service Management Section**
  Allows the user to add, edit, and view service records for the selected motorcycle.

User actions such as button clicks and list selections are handled through event-driven methods, which then call the appropriate methods in the application logic layer.

3.3 Application Flow

The overall flow of the application can be described as follows:

1. The application starts by initializing the main window and creating a User object along with a MotoCareApp instance.

2. The user enters motorcycle data through the input form and adds a new motorcycle to the system.
3. The motorcycle is displayed in the motorcycle list section of the interface.
4. When a motorcycle is selected, the service history associated with that motorcycle is displayed.
5. The user can add a new service record by filling in the service form and submitting the data.
6. If a service record contains incorrect information, the user can select the record, edit the data, and update it.
7. The user can also remove a motorcycle, which automatically removes its associated service records from the view.
8. After each operation, the interface is refreshed to reflect the updated data.

This flow ensures that all user interactions are processed systematically and that the displayed information remains consistent with the underlying data.

3.4 Error Handling and Input Validation

To ensure application stability, input validation is applied before data is processed. The system checks for required fields, validates numeric input such as mileage and cost, and enforces a fixed date format for service records.

Error messages are displayed using dialog boxes to inform the user when invalid input is detected. This prevents incorrect data from being stored and improves the overall user experience.

CHAPTER IV
IMPLEMENTATION OF OBJECT-ORIENTED PROGRAMMING CONCEPTS

4.1 Overview of Object-Oriented Design

he MotoCare application applies Object-Oriented Programming (OOP) concepts directly in its source code. Each concept is implemented through specific classes, methods, and design decisions. This chapter explains how encapsulation, inheritance, polymorphism, and abstraction are realized in the actual implementation of the system.

4.2 Encapsulation

Encapsulation is implemented by restricting direct access to class attributes and providing controlled interaction through methods and properties.

For example, in the Motorcycle class (motorcycle.py), internal attributes such as mileage and service history are defined as protected:

```
self._current_km = current_km
self._service_records: list[ServiceRecord] = []
```

FIGURE 4.1 ENCAPSULATION

These attributes are not accessed directly by the user interface. Instead, service records are added through the add_service() method:

```
def add_service(self, record: ServiceRecord):
    self._service_records.append(record)
    self._current_km = record.km
```

FIGURE 4.2 ENCAPSULATION

This ensures that whenever a service record is added, the motorcycle's mileage is automatically updated, preserving data consistency.

4.3 Inheritance

Inheritance is applied by defining a base class Vehicle and a subclass Motorcycle.

In vehicle.py, the Vehicle class is declared as an abstract class:

```python
class Vehicle(ABC):
    def __init__(self, plate, brand, model, year):
        self._plate = plate
        self._brand = brand
        self._model = model
        self._year = year

    @abstractmethod
    def get_info(self):
        pass
```

FIGURE 4.3 ENCAPSULATION

The Motorcycle class inherits from Vehicle and extends its functionality:

```python
class Motorcycle(Vehicle):
    def __init__(self, plate, brand, model, year, current_km):
        super().__init__(plate, brand, model, year)
        self._current_km = current_km
```

FIGURE 4.4 ENCAPSULATION

Through inheritance, common vehicle attributes are reused without duplication, while motorcycle-specific behavior is implemented in the subclass.

4.4 Polymorphism

Polymorphism is demonstrated through method overriding. The get_info() method is declared in the Vehicle class and overridden in the Motorcycle class:

```python
def get_info(self):
    return f"[{self._plate}] {self._brand} {self._model} ({self._year}) - {self._current_km} km"
```

FIGURE 4.5 POLYMORPHISM

Although the method name is the same, the implementation in Motorcycle provides a more detailed output. This allows objects derived from Vehicle to respond differently to the same method call.

4.5 Abstraction

Abstraction is implemented by using the ABC module and the @abstractmethod decorator in the Vehicle class. The abstract method get_info() enforces a common interface for all subclasses:

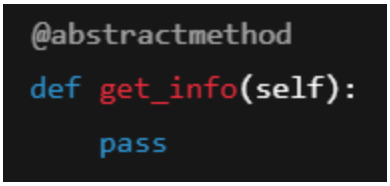

FIGURE 4.6 ABSTRACTION

This prevents the Vehicle class from being instantiated directly and ensures that every subclass provides its own implementation of the method.

4.6 Benefit

By applying OOP concepts directly in the code, the MotoCare application achieves a clean and maintainable structure. Each class has a clear responsibility, and changes in one part of the system do not significantly affect other components. This approach also makes the application easier to extend, for example by adding new vehicle types or additional features.

CHAPTER V
INTERFACE SCREENSHOTS AND EXPLANATION

5.1 Main Application Interface

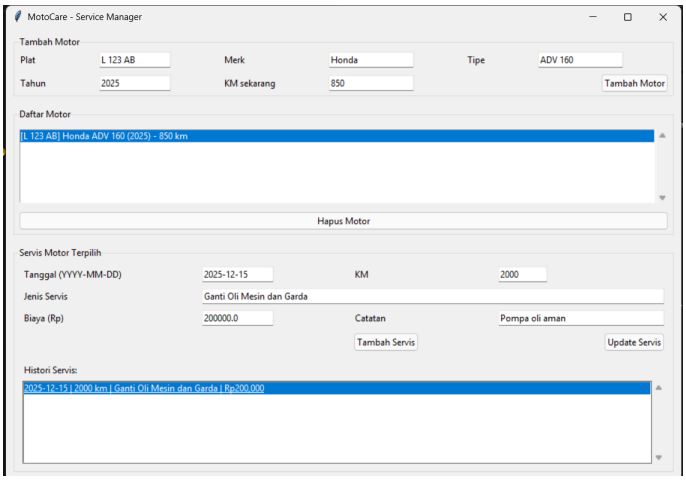

FIGURE 5.1 MAIN APPLICATION INTERFACE

Figure 5.1 shows the main interface of the MotoCare application. This interface serves as the central point for managing motorcycles and their service records. The interface is divided into three main sections:

- Motorcycle input section
- Motorcycle list and management section
- Service record management section

This layout is designed to provide a clear workflow from motorcycle registration to service history management.

5.2 Add Motorcycle Section

The upper part of the interface is the **Add Motorcycle** section. This form allows users to register a new motorcycle into the system. The input fields include:

- **License Plate**: unique identifier for each motorcycle
- **Brand**: manufacturer of the motorcycle
- **Type**: motorcycle model
- **Year**: production year
- **Current KM**: current mileage of the motorcycle

After all required fields are filled, the user can click the **Add Motorcycle** button to store the data. The motorcycle will then appear in the motorcycle list below. This feature ensures that each motorcycle is stored as a Motorcycle object in the application.

5.3 Motorcycle List and Management

The **Motorcycle List** displays all motorcycles registered by the user. Each item shows essential information such as license plate, brand, model, year, and current mileage.
The user can:

- Select a motorcycle to view or manage its service history
- Remove a motorcycle using the **Delete Motorcycle** button

When a motorcycle is selected, the system automatically updates the service section to reflect the selected motorcycle. This prevents service data from being assigned to the wrong vehicle.

5.4 Service Record Input Section

The **Selected Motorcycle Service** section is used to add or update service records for the selected motorcycle. The input fields include:

- **Service Date**
- **Mileage at Service**
- **Service Type**
- **Service Cost**
- **Notes**

By clicking the **Add Service** button, a new ServiceRecord object is created and attached to the selected motorcycle.
The **Update Service** button allows users to edit an existing service record, which is useful for correcting input errors. This functionality demonstrates dynamic object manipulation in the application.

5.5 Service History Display

The **Service History** section displays a chronological list of all service records associated with the selected motorcycle.

Each entry shows:

- Service date
- Mileage
- Service type
- Cost

When a service record is selected, its details are automatically loaded into the input fields above, allowing easy editing. This design improves usability and ensures data accuracy.

CHAPTER 6
CONCLUSION

Based on the results of the design and implementation of the MotoCare application, it can be concluded that the application has been successfully developed as an Object-Oriented Programming (OOP)-based system for managing motorcycle data and service history. The application is able to assist users in recording and managing vehicle data as well as maintenance history in a structured and systematic manner.

The implementation of OOP concepts such as encapsulation, inheritance, polymorphism, and abstraction is clearly demonstrated through the design of the Vehicle, Motorcycle, ServiceRecord, User, and MotoCareApp classes. The clear division of responsibilities within each class results in a modular code structure that is easy to understand and maintain, as well as easy to extend.

In addition, the use of a class diagram helps visualize object relationships and supports a well-planned system implementation process. The graphical user interface developed using Tkinter is also designed to be simple and user-friendly, making it easier for users to add, update, and manage motorcycle data and service history.

Overall, the MotoCare application has fulfilled the learning objectives of the Object-Oriented Programming course and the requirements of the Final Examination (UAS), while also serving as a practical example of applying OOP concepts in software development.

APPENDIX

Repository Links: https://github.com/Richie-Corneliu/Motocare-App