< Fines Payment Manager >
Analysis and Design Document

Student: Curac Mihai - Ionuț
Group: 30431

Revision History

Date	Version	Description	Author
<11/apr/23>	<0.5>	Domain Model, Architectural Design, Component and Deployment diagrams	<Curac Mihai - Ionut>
<2/may/23>	<1.0>	Design Model, Data Model	<Curac Mihai - Ionut>
<24/may/23>	<1.5>	Final additions	<Curac Mihai - Ionut>

Table of Contents

I. Project Specification

The specification for the project is to design and implement a client-server application for managing the payment of traffic fines. The application consists of two components which can be accessed using a username and a password. One component is used by the post office employees to register the persons that have paid a traffic fine. The other component is used by the police employees to do CRUD operations on the drivers' information (name, address, driving license details, identity card details), and on the fines, as well as to create reports. When a post office employee registers a person that has paid a traffic fine, the application notifies the police employees from the corresponding police station. All the information about users, drivers and fines is stored in a database.

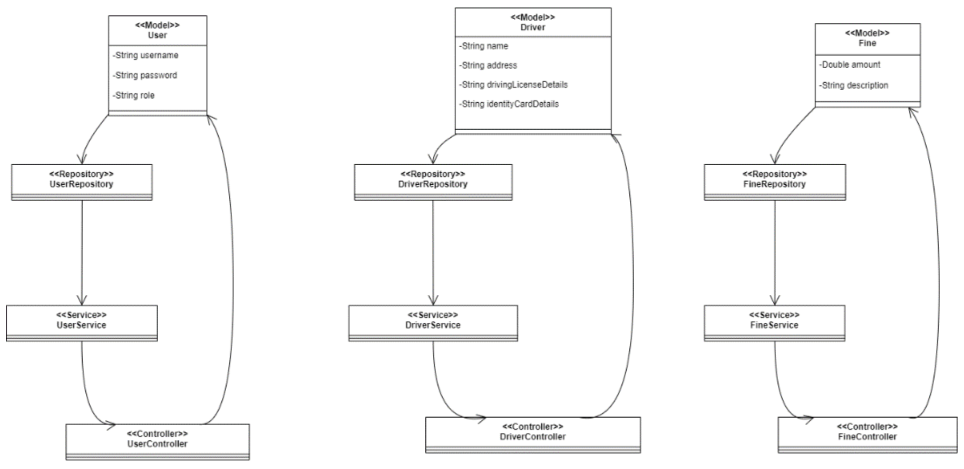
II. Elaboration – Iteration 1.1

1. Domain Model

The domain model for the traffic fines payment management system consists of the following classes:

- User: Represents a user of the system, who can be either a post office employee or a police employee. Each user has a username and a password for authentication purposes.
- Driver: Represents a driver who has received a traffic fine. Each driver has a name, an address, driving license details, and identity card details.
- Fine: Represents a traffic fine that has been issued to a driver. Each fine has a unique identifier, a driver associated with it, an amount, a date of issue, and a date by which it must be paid.
- PoliceStation: Represents a police station that employs police employees who use the system to manage drivers' information, fines, and create reports. It has a unique identifier, a name, an address, and a phone number.
- PostOffice: Represents a post office that employs post office employees who use the system to register the persons that have paid a traffic fine

The conceptual class diagram of the system is shown below:



2. Architectural Design

2.1 Conceptual Architecture

The fines payment management system will use a client-server architecture with a RESTful web service as the communication mechanism between the client and the server. The client will be a web-based user interface that can be accessed from any device with a web browser. The server will be a Java Spring Boot application that will use Hibernate to access the database.

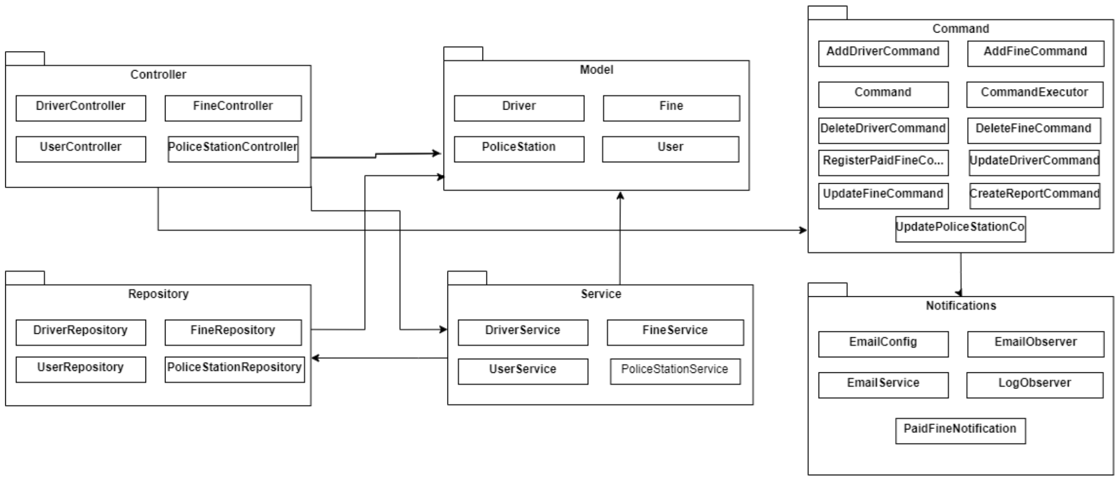
The server-side architecture will be based on the Model-View-Controller (MVC) pattern. This pattern separates the application into three components: the model, the view, and the controller. The model represents the data of the application, which can be a single object or a collection of objects. The view represents the provided information in a particular format. Finally, the controller contains the business logic of an application. It acts as an intermediary between the model and the view, handling user input and updating the model accordingly.

The RESTful web service will be implemented using the Spring MVC framework, which provides built-in support for creating RESTful web services. The service will expose RESTful endpoints for performing CRUD operations on drivers, fines, and users. These endpoints will accept and return JSON data.

Overall, the use of the client-server architecture with a RESTful web service and the MVC pattern provides a scalable, maintainable, and flexible solution for the fines payment management system.

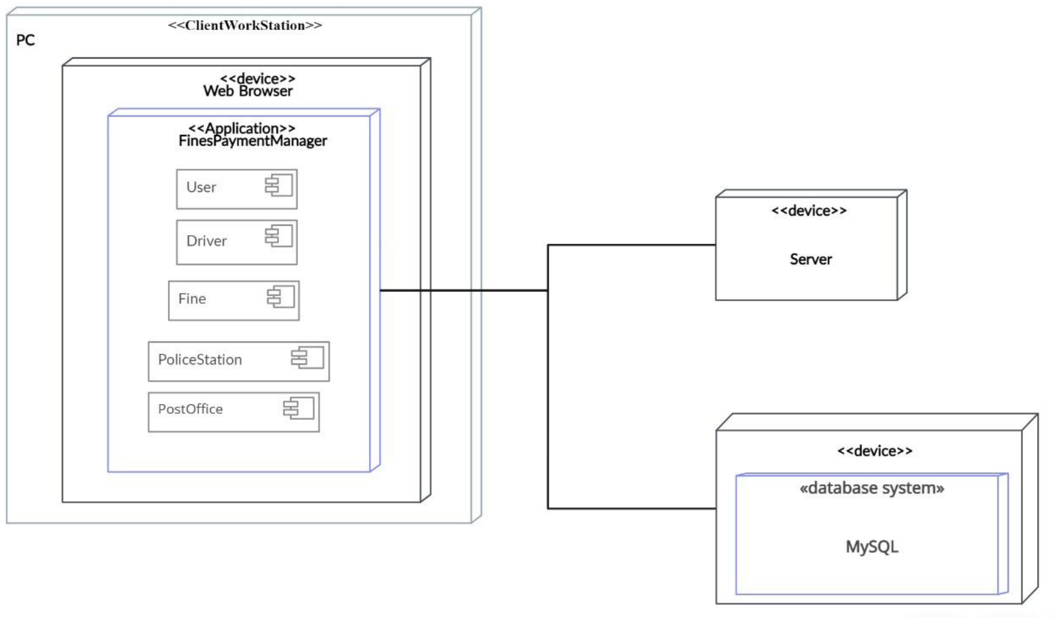
2.2 Package Design

The packages diagram is the following:

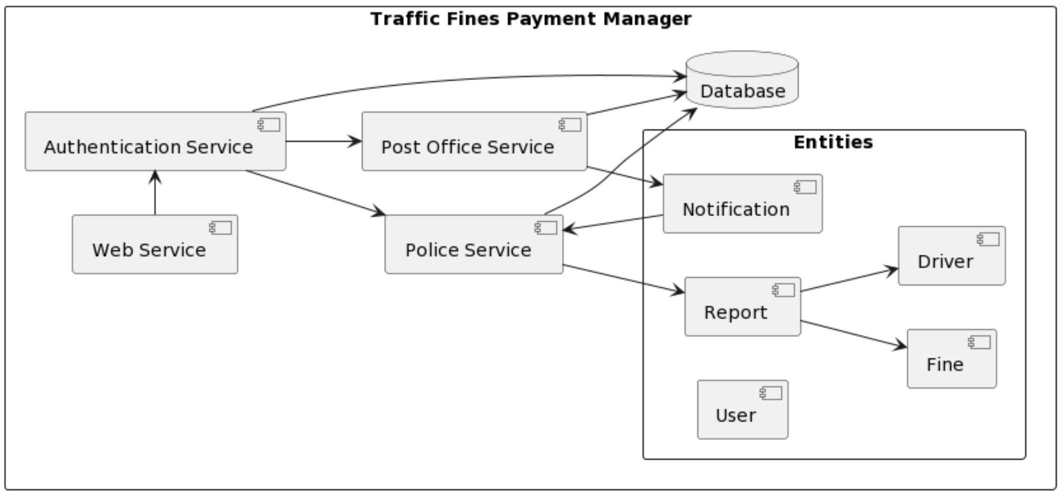


2.3 Component and Deployment Diagrams

The deployment diagram for the Fines Payment Manager is the following:



The component diagram is illustrated below:

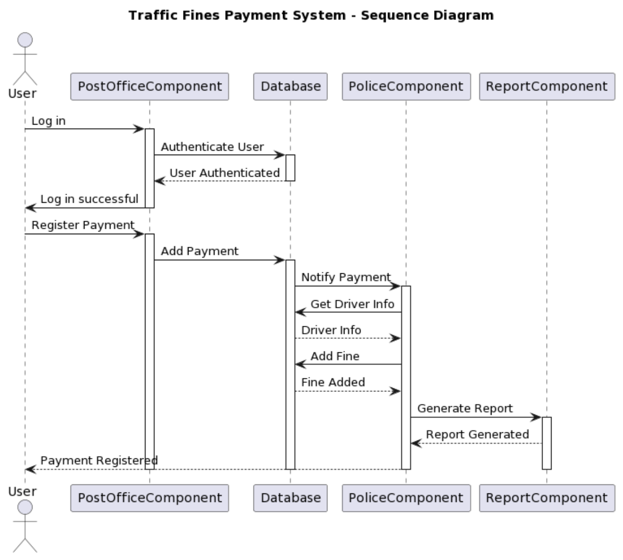


III. Elaboration – Iteration 1.2

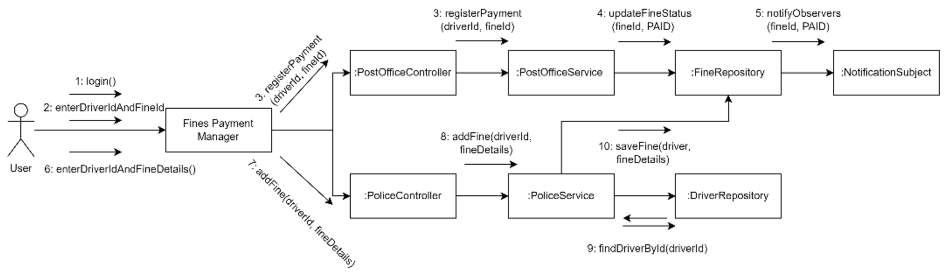
1. Design Model

1.1 Dynamic Behavior

The sequence diagram for the scenarios of adding a fine and registering a payment is illustrated below:

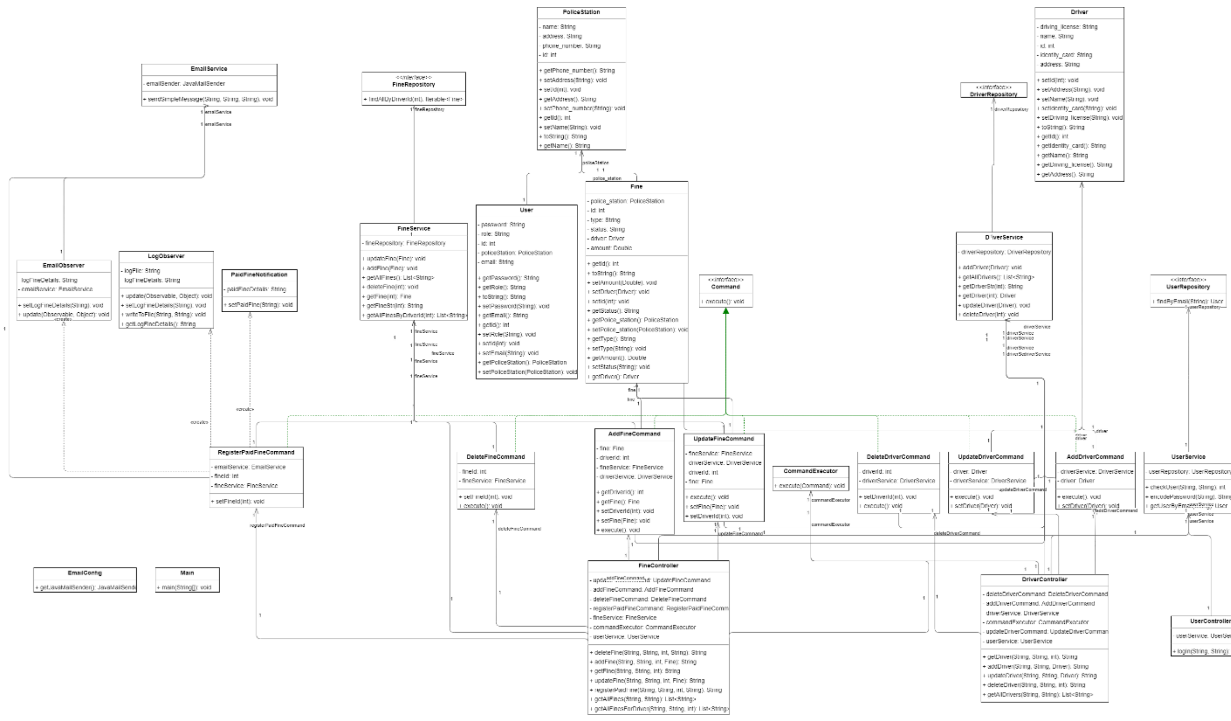


The communication diagram for the same scenarios is illustrated below:


1: login()
2: enterDriverIdAndFineId
6: enterDriverIdAndFineDetails()
User
Fines Payment Manager
3: registerPayment (driverId, fineId)
7: addFine(driverId, fineDetails)
:PostOfficeController
:PostOfficeService
:FineRepository
:NotificationSubject
3: registerPayment (driverId, fineId)
4: updateFineStatus (fineId, PAID)
5: notifyObservers (fineId, PAID)
:PoliceController
:PoliceService
:DriverRepository
8: addFine(driverId, fineDetails)
10: saveFine(driver, fineDetails)
9: findDriverById(driverId)

1.2 Class Design

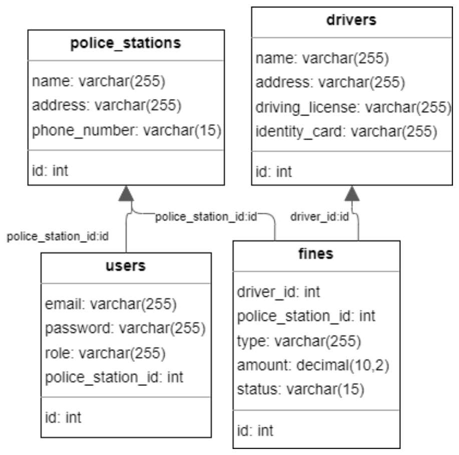
The UML class diagram of the Fines Payment Manager is the following:



2. Data Model

The chosen system's data model includes the following entities:

- Driver Entity:
    - Represents a driver in the system.
    - Has attributes such as id, name, address, driving license, and identity card.
    - Has a one-to-many relationship with fines (a driver can have multiple fines).
- Fine Entity:
    - Represents a traffic fine in the system.
    - Has attributes such as id, type, amount, and status.
    - Belongs to a driver (many fines belong to one driver) and a police station (many fines belong to one police station).
- User Entity:
    - Represents a user in the system.
    - Has attributes such as id, email, password, and role.
    - Belongs to a police station (many users belong to one police station).
- PoliceStation Entity:
    - Represents a police station in the system.
    - Has attributes such as id, name, address, and phone number.
    - Has a one-to-many relationship with users (a police station can have multiple users) and fines (a police station can issue multiple fines).



This data model allows for storing and retrieving information about users, drivers, fines, and police stations in a structured and normalized way. It also supports various operations that are required by the system's functionality, such as registering payments, adding fines, creating reports, and sending notifications.

3. Unit Testing

Unit testing was used as a testing strategy for the fines payment manager. The chosen unit tests focus on testing individual components of the system, such as controllers and services, in isolation. More precisely, for the FineControllerTest class the tests included were when “Add Fine With Valid Credentials And Valid Fine then Return Success Message” and “when Add Fine With Invalid Login then Return Login Failed Message”, whereas for the DriverControllerTest class the tests done were “when Get Driver then Return Driver”, “when Get Driver With Invalid Login then Return Login Failed Message”, and “when Get Driver With Non Existing Id then Return Not Found Message”. Validation testing was also used to validate the correctness of user input and ensure that the system can handle unexpected or invalid input. Tests were given through Postman HTTP Requests.

IV. Elaboration – Iteration 2

1. Architectural Design Refinement

The architectural design has been refined to improve the overall structure and organization of the system. The changes made are as follows:

- Conceptual Architecture: The conceptual architecture remains the same, following the client-server model with a RESTful Spring Boot application. However, the package design and component diagram have been updated to reflect the changes in the system. Mainly, the PoliceStation component was added as a bridge between fines and police employees.
- Package Design: The package design follows package design principles to ensure modularity, cohesion, and low coupling. The packages are organized as follows:
    - Controller - Contains the controllers responsible for handling HTTP requests and responses.
    - Model - Contains the entity classes representing the data models used in the system.
    - Service - Contains the service classes that implement the business logic and interact with the repositories.
    - Repository - Contains the repository interfaces and their implementations for database access.
    - Command - Contains the command classes implementing the Command pattern for handling specific actions.
    - Notifications - Contains the classes implementing the Observer pattern for sending notifications.

2. Design Model Refinement

The UML class diagram has been refined to improve the design of the classes and apply class design principles and GRASP. The PoliceStation class was added as mentioned above, as well as the reports functionality that is available for police employees. The class that implements the fines report functionality is CreateReportCommand, part of the Command design pattern.

V. Construction and Transition

1. System Testing

The testing methods include Data Flow Testing, since the unit tests described above cover different scenarios by manipulating the flow of data through the system, and Partitioning, because the tests cover different partitions of input data, such as valid and invalid login credentials, existing and non-existing driver IDs, etc. Moreover, Integration Testing was used to test how the components of the system interact with each other. Integration tests were written to verify that the system components are integrated correctly and working together as intended.

2. Future improvements

While the current system meets the specified requirements, there are several areas that could be improved in future iterations:

- Enhanced Security: Implement additional security measures such as encryption for sensitive data, role-based access control, and user authentication mechanisms.
- Logging and Error Handling: Improve the logging and error handling mechanisms to provide more detailed error messages and better traceability of system events.
- Performance Optimization: Optimize the system's performance by implementing caching mechanisms, database optimizations, and load balancing techniques.

VI. Bibliography

- https://refactoring.guru/design-patterns/command
- https://www.baeldung.com/java-command-pattern
- https://www.baeldung.com/java-observer-pattern
- https://www.baeldung.com/java-write-to-file
- https://www.baeldung.com/spring-email
- https://www.baeldung.com/spring-boot-testing
- https://spring.io/guides/gs/testing-web/