A Technical Report on the Implementation of TransmiApp

David Felipe García León
Juan Esteban Bedoya Lautero

System Engineering

Universidad Distrital Francisco José de Caldas

Contents

A Technical Report on the Implementation of TransmiApp

***Abstract*—This document presents the design and implementation of an application that manages TransMilenio routes and stations, using an architecture based on design patterns to enhance system scalability and maintainability. The software structure is described, including UML diagrams that represent the interaction between different components and how SOLID principles are applied in the architecture.**
Additionally, the functional and non-functional requirements are detailed, along with the user stories that guided the system's development. The design patterns used are analyzed, including Chain of Responsibility for recharge validation, Strategy for route planning, and Template for code reuse.
The tests conducted verify the system's efficiency in terms of response time and accuracy in route and station searches, demonstrating the effectiveness of the architectural decisions made. Finally, conclusions are presented regarding the impact of design patterns on the proposed solution, along with recommendations for future improvements.

I. Introduction

El crecimiento acelerado de las ciudades ha generado una mayor demanda de sistemas de transporte eficientes. En este contexto, TransMilenio, como sistema de transporte masivo en Bogotá, enfrenta desafíos relacionados con la optimización de rutas, la gestión de estaciones y la experiencia del usuario. Una de las principales problemáticas es la falta de herramientas tecnológicas que permitan a los usuarios acceder fácilmente a información relevante sobre sus trayectos, recargar sus tarjetas y gestionar su movilidad de manera eficiente.

Este trabajo presenta el desarrollo de una aplicación diseñada para abordar estos problemas mediante un sistema de gestión de rutas y estaciones basado en principios de ingeniería de software. La solución propuesta se apoya en patrones de diseño para garantizar modularidad, escalabilidad y mantenibilidad. Además, se han aplicado principios SOLID en la arquitectura del sistema con el fin de mejorar la organización del código y facilitar su evolución.

El documento detalla los requisitos funcionales y no funcionales, las historias de usuario que guiaron el desarrollo, así como los principales diagramas UML que representan la estructura e interacción de los componentes del sistema. También se analiza la implementación de patrones de diseño como Chain of Responsibility para la validación de recargas y Strategy para la planificación de rutas, demostrando su impacto en la solución.

Finalmente, se presentan los resultados obtenidos tras la implementación del sistema, junto con un análisis de desempeño y una discusión sobre posibles mejoras futuras.

II. User Histories

The following user stories reflect the needs of users regarding the key functionalities implemented in the TransMilenio management application.

Station and Stop Search

1. Search for nearby stations and routes
   As a new user of the application, I want to search for stations near my current location so that I can quickly identify the most convenient station for my trip.

2. View detailed station information
   As a frequent user, I want to access detailed information about each station, including available routes, schedules, and connections to other transport systems, so that I can better plan my trips.

Trip Planning

3. Optimal route suggestions
   As a user who relies on TransMilenio for my daily commute, I want the app to suggest the

best route based on my origin and destination so that I can minimize my travel time.

4. View schedules and estimated arrival times
As a user who plans trips in advance, I want to see the operating schedules of each route and the estimated arrival times of buses so that I can better organize my itinerary.

5. Consideration of intermediate stops in route planning
As a user who needs to transfer between buses, I want the app to show if I need to change stations or take another route so that I can avoid confusion during my trip.

User Management

6. User registration and login
As an app user, I want to register with my personal data and securely log into my account so that I can manage my preferences and transportation balance.

7. Manage personal information and preferences
As a registered user, I want to update my personal data and configure notification preferences so that I can personalize my experience within the app.

Balance and Card Recharge Management

8. Check real-time card balance
As a frequent TransMilenio user, I want to check my transportation card balance in the app so that I know if I need to recharge before traveling.

9. Recharge transportation card
As a user who needs to recharge frequently, I want to top up my card directly through the app and receive instant confirmation so that I can avoid issues with delayed payments.

III. Functional and non-functional requirements.

Functional:

- User login: User login capability to save data (card number for recharges).
- Search for buses according to their station: With the search for each station, show the buses that stop at that station.
- See the route of each bus: Looking for the TransMilenio bus, see the stops it makes along its route.
- Trip planning: Introducing a station "A" and a station "B" or any of the favorite stations to show which route or routes are more efficient.
- System card recharge: With the card previously stored in the user's independent data, a virtual recharge can be made.

Non-functional:

- Maintainability and Modularity: The code strictly follows the SOLID principles to maintain a clean and easy-to-maintain application.
- Five different patterns are used, which help us improve our code reuse, keep sequences organized, and store important data as objects.
- The documentation of our code is done 100% with Javadoc in Java and docstrings in Python.
- There is a standard format in our JSON files for route and station information.
- Technological Restrictions: Our application does not use external technologies, it only requires our private JSON files to function.
- The application works on Windows and Linux without the need for complex additional configurations.

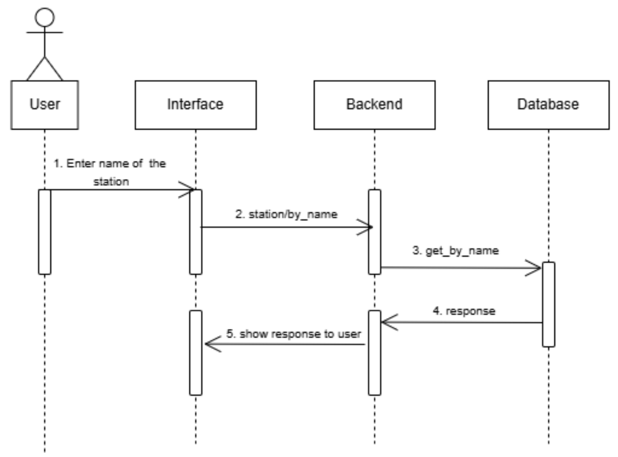
IV. Technical considerations

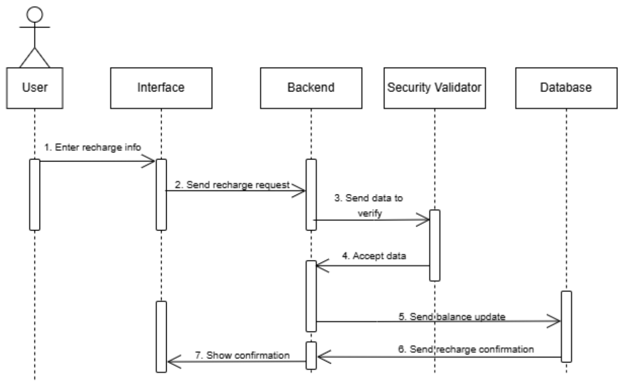
The implementation of this project will leverage a dual-backend architecture, utilizing both Java and Python to ensure modularity and scalability. The Java component, developed using Java 17, will handle user management and maintain a history of user searches by persisting data in JSON files, with the records being associated through unique user IDs. This design facilitates straightforward data persistence and retrieval.

Conversely, the Python component, built with Python 3.10, will interface with JSON files to retrieve data regarding Transmilenio stations and routes. FastAPI will be employed to serve API endpoints, enabling efficient querying of routes or stations by name. Supplementary libraries, including python-dotenv for managing environment variables, uvicorn as an ASGI server, and pydantic for data validation, will be integrated to enhance the overall robustness and maintainability of the application.

Furthermore, modern development tools such as Docker for containerization, Postman for API testing, and GitHub for version control will be utilized to streamline the development process and ensure a resilient deployment pipeline.

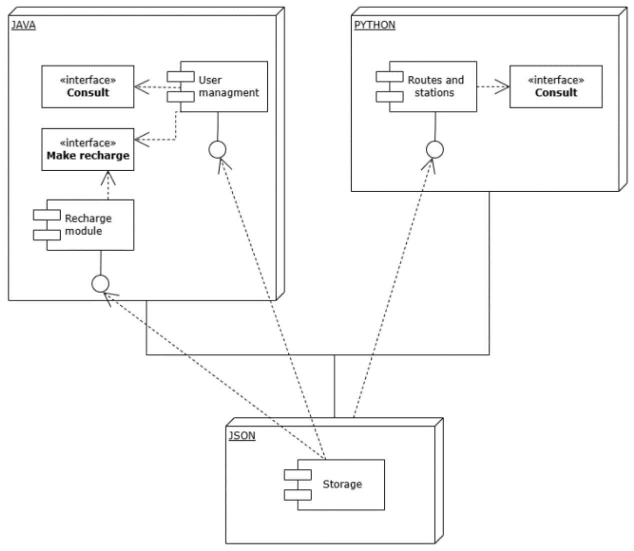
V. Sequence Diagrams.



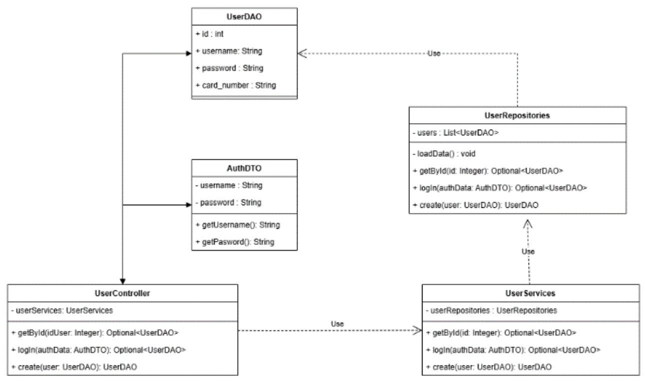


VI. Architecture: Including Component Diagram and Class Diagrams

1. Componen Diagram:



2. Class Diagrams:


UserDAO
+ id : int
+ username: String
+ password : String
+ card_number : String
AuthDTO
- username : String
- password : String
+ getUsername(): String
+ getPasword(): String
UserRepositories
- users : List<UserDAO>
- loadData() : void
+ getById(id: Integer): Optional<UserDAO>
+ logIn(authData: AuthDTO): Optional<UserDAO>
+ create(user: UserDAO): UserDAO
UserController
- userServices: UserServices
+ getById(idUser: Integer): Optional<UserDAO>
+ logIn(authData: AuthDTO): Optional<UserDAO>
+ create(user: UserDAO): UserDAO
UserServices
- userRepositories : UserRepositories
+ getById(id: Integer): Optional<UserDAO>
+ logIn(authData: AuthDTO): Optional<UserDAO>
+ create(user: UserDAO): UserDAO
Use
Use
Use

VII. Analysis of SOLID Principles in the Architecture

Single Responsibility Principle (SRP)

Separation of concerns in the services/ and controllers/ directories, ensuring that services handle business logic while controllers manage request handling.

Open/Closed Principle (OCP)

Implementation of the Strategy Pattern for route planning, allowing new strategies to be added without modifying existing code.
Use of interfaces for behavior definition, promoting extensibility.

Liskov Substitution Principle (LSP)

Usage of interfaces to ensure that subclasses can replace their parent class without affecting functionality.

Interface Segregation Principle (ISP)

Specialized interfaces for different services, such as user management and balance recharge, ensuring that each module only depends on the methods it actually needs.

Dependency Inversion Principle (DIP)

Dependency injection in services to reduce tight coupling.
Implementation of the Chain of Responsibility Pattern in balance validation to decouple different validation steps.

VIII. Analysis and Details of Design Pattern Implementation

Implemented Patterns

Template Method (Behavioral Pattern)

Implementation: Used in the route and station search functionality.
Justification: Establishes a standard structure for search processes while allowing subclasses to implement specific variations.

Strategy (Behavioral Pattern)

Implementation: Applied in the route planning system.
Justification: Enables the selection of different routing algorithms based on user preferences or real-time traffic conditions, enhancing flexibility and modularity.

Chain of Responsibility (Behavioral Pattern)

Implementation: Used in the validation process for card balance recharge.
Justification: Facilitates sequential validation steps, such as card validity, balance limits, and transaction security, without creating strong dependencies between components.
Patterns Not Implemented and Their Justifications

Creational Patterns

Builder

Not necessary, as the project does not involve constructing complex objects with multiple configurations.

Factory Method

Not required, since object creation is straightforward and does not require dynamic instantiation based on varying conditions.

Abstract Factory

Typically used to manage families of related objects, which is not a requirement in this project.

Singleton

While it could be useful for managing shared resources such as database connections, there is no explicit Singleton implementation in the project.

Prototype

Object cloning is not a requirement, making this pattern unnecessary.
Structural Patterns

Bridge

Not required, as the project does not involve dynamically separating abstraction from implementation.

Composite

Unnecessary because the system does not handle hierarchical structures where individual and composite objects need to be treated uniformly.

Proxy

While useful for access control or lazy loading, this pattern was not necessary for the implemented functionality.

Flyweight

Not used, as memory optimization for a large number of similar objects was not a priority.

Decorator

While it could provide dynamic functionality extension, the existing approach does not require this level of flexibility.

Adapter

Typically used for integrating external systems, but since the project does not directly interact with third-party APIs, its implementation was not necessary.

Facade

Could simplify interactions between complex subsystems, but the current architecture does not require an additional abstraction layer.
Behavioral Patterns

Iterator

Collection traversal is handled using built-in iteration mechanisms, eliminating the need for an explicit Iterator pattern.

Memento

Not required, as the system does not include functionality for restoring previous states.

State

While useful for managing dynamic workflow changes, the system does not require this structure.

Mediator

Not necessary, as direct communication between components suffices for the current implementation.

Command

Could be beneficial for undo/redo operations or request queuing, but it was not explicitly implemented.

Observer

While suitable for event-driven interactions, no explicit implementation of the Observer pattern was identified in the system.

IX. Analysis of Best Programming Practices and Anti-Patterns

Separation of Concerns (SoC):

The project follows a clear division of responsibilities between different modules, ensuring that each component has a well-defined role. The separation between route search, travel planning, user management, and card balance operations adheres to this principle, improving maintainability and scalability.

Encapsulation and Information Hiding:

Data and business logic are encapsulated within their respective classes, preventing direct access to internal states and enforcing controlled interaction through getter and setter methods. This enhances code modularity and security.

Use of Design Patterns:

The implementation of Template Method, Strategy, and Chain of Responsibility follows established software engineering principles to improve flexibility and extensibility. These patterns reduce code duplication and facilitate future modifications.

Error Handling and Exception Management:

The system incorporates structured exception handling mechanisms to prevent crashes and ensure proper logging of errors. This improves system robustness by allowing controlled failure responses.

Code Readability and Maintainability:

The project follows consistent naming conventions, meaningful variable and method names, and structured documentation, improving readability and facilitating future modifications.

Version Control and Collaboration:

The use of Git and a structured branching model ensures efficient collaboration, version control, and rollback capabilities when necessary.
Identified and Avoided Antipatterns

God Object (Avoided):

The project avoids concentrating excessive responsibilities in a single class. Instead, responsibilities are distributed among specialized classes following the Single Responsibility Principle (SRP).

Spaghetti Code (Avoided):

The code structure follows modularity principles, reducing unnecessary dependencies and improving comprehension, thereby avoiding convoluted and interdependent logic.

Hardcoded Values (Partially Present):

While most configurations are parameterized, some static values exist within the codebase. Future improvements should focus on externalizing all configurations into environment variables or configuration files.

Duplicate Code (Minimized through Template Method):

The Template Method pattern significantly reduces code duplication in route and station searches. However, some minor redundancies could still be refactored for further optimization.

Lack of Documentation (Partially Present):

Although the code is structured and follows best practices, certain sections lack detailed documentation. Enhancing inline comments and API documentation would further improve maintainability.

Excessive Coupling (Avoided through Dependency Injection):

The project minimizes tight coupling by leveraging dependency injection principles, facilitating scalability and testing.

Conclusion

X. Conclusions

The development of this project demonstrates the importance of applying software engineering principles to create a scalable, maintainable, and efficient system. By utilizing design patterns such as Template Method, Strategy, and Chain of Responsibility, the architecture was structured to enhance modularity and code reusability while ensuring compliance with SOLID principles.

The implementation of good programming practices, including encapsulation, separation of concerns, and structured exception handling, has contributed to the system's robustness. Additionally, avoiding common antipatterns such as God Object and Spaghetti Code has improved code clarity and maintainability.

Overall, this project highlights the effectiveness of a well-structured software design in addressing complex challenges within public transportation systems. It demonstrates how applying design patterns and best practices leads to a reliable and adaptable solution capable of enhancing user experience and operational efficiency.